Atmel Nantes S.A.
BP 70602 – La Chantrerie
44306 Nantes cedex 3 – France
Tél : +33 (0)2.40.18.18.18
www.atmel.com

Blaise LENGRAND
blaise.lengrand@gmail.com
Promotion WATT

4, rue Merlet de la Boulaye BP30926
49009 Angers cedex 01 – France
Tél : +33 (0)2.41.86.67.67
www.eseo.fr

DSP library for the AVR32UC3 - report	**Type**: End of studies training report **Period**: From February 5th till July 27th, 2007 **File**: RapportDeStage.doc **Author**: Blaise LENGRAND **Language**: English **Created**: 5/28/2007 **Last modification**: 6/25/2007

Destination: Cyrille Boulanger, Daniel Genêt.

Digital Signal Processing library optimized for the avr32uc3

Atmel Nantes S.A.

AVR32

TABLE OF CONTENTS

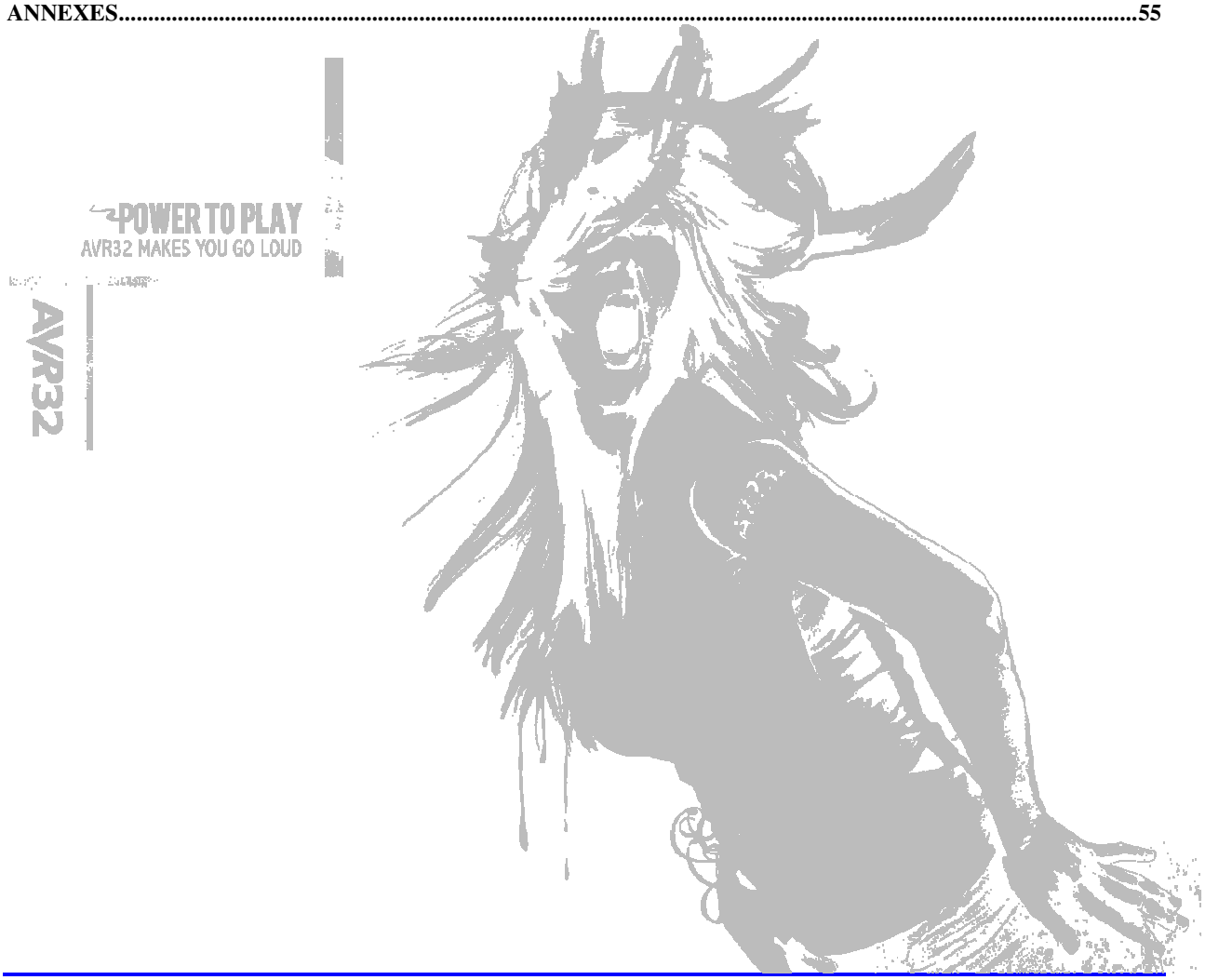


TABLE OF FIGURES

Internship summary

Internship summary

Atmel Nantes S.A.
La Chantrerie
BP70602
44306 Nantes cedex 3
France
Tel.: (33) 2 40 18 18 18
Fax.: (33) 2 40 18 19 20
http://www.atmel.com

Blaise LENGRAND
blaise.lengrand@gmail.com
End of studies' trainee
Promotion WATT
February – July 2007

Digital Signal Processing Library

1. Subject

With the development of a new AVR32 microcontroller, Atmel decided to include in their products more software solutions in order to enhance and simplify applications' development. This new product has interesting features such as DSP instructions which makes it an ideal solution for telephony and other small digital signal processing applications. In order to turn to good account those instructions, Cyrille Boulanger offered an internship for the development of an optimized digital signal processing library for this new product: the AVR32UC3.

2. Results

The library provides not only basic DSP functions such as filtering, windowing or transforms but also vector management functions, fixed-point number operators, signal generation routines and a complete set of debugging functions. It also provides another sub-library designed for more specifics applications such as audio processing and is based in the first library. All of these functions have been tested and profiled and are all supported by Atmel's customer support. This library is still in development but a first release will be available in July.

Acknowledgements

With this report, I would like to thank my internship's master, Cyrille Boulanger, without whom this experience would not have been possible. He supervised all of my work during this internship and helped me make the right decisions in order to design the library.

I also would like to thank all the members of the Application Laboratory which together makes a great team, that's agreeable to work with. Thank you specifically to Benoit Thebaudeau, an essential member of this group, for having taken the time to help me on several problems mainly due to poor configurations. Thank you also to Stéphane Mainchain for having helped me send "Bugzillas" on the avr32software network. Thank you to Regis Latawiec, the marketer of the avr32uc3 product line, for having taken time to advise me during the specification of the library.

Thank you also to the Sodexho team for those delightful meals they prepared.

I also would like to thank to ESEO (Ecole Superieure d'Electronique de l'Ouest) for all the knowledge it gave me and mainly Mr. Genet for having followed my internship and for its interesting classes. Thank you also Mr. Perdriau for having traveled from Angers to listen to my advancement report in June.

Resume (French)

blaise.lengrand@gmail.com
06.30.25.35.00

Blaise LENGRAND

Allée des Phoënix, Lieu-dit Keranouat
29910 Trégunc
Détenteur du permis B

Assistant Ingénieur,
France entière, Février – Juin

EDUCATION

2004 – Présent 1ère, 2ème et 3ème année en école d'ingénieur à l'E.S.E.O., Ecole Supérieur d'Electronique de l'Ouest, conférant un diplôme d'ingénieur généraliste spécialisé dans les domaines de l'électronique, l'informatique et les télécommunications. Diplôme attendu en Juin 2007.

2002 – 2004 Classes préparatoires à l'E.S.E.O.

LANGUES

Français Langue maternelle.
Anglais: Bon niveau (TOEFL 557, First Certificate of Cambridge niveau C).
Espagnol: Bon niveau (étudié pendant 9 ans).
Allemand: Connaissances de base (étudié pendant 5 ans).

COMPETENCES EN INFORMATIQUE

Général:

Systèmes: Linux, Microsoft Windows 95/98/NT/2000/XP.
Logiciels: Microsoft Word, Excel, Visio…

Langages de programmation:

C: UNIX, LKMs, Microsoft Windows API.
Assembleur: INTEL x86, Microchip MPASM, SATURN et Motorola.
Objet: C++, Java, UML, XML. **IDE**: Visual Studio 6/.net/2005, Eclipse 3.0, Dev-C++.
Web: HTML, Javascript, PHP, CSS, Flash, SQL, ASP.

Cryptographie:

Chiffrement symétrique: RC5.
Chiffrement asymétrique: RSA, Diffie-Hellman.
Fonctions de hachage: MD5, SHA1.

Algorithmique:

I.A.: Réseaux de neurones.
Recherche du plus court chemin: A*, Dijkstra.
Compression: JPEG, Huffman, LZW, RLC.

Autres:

Formats: P.E., FAT, ZIP, BMP, WAV.

COMPETENCES EN ELECTRONIQUE

Général:

Circuits: Conception, tests, routage et réalisation.
- *ADS, Eagle, Pspice, Microcap.*

Numérique:

Microprocesseurs/Microcontrôleurs: Intel x86, Microchip PIC, Motorola 68K, Power PC, DSP.
- *Microchip MPLAB IDE, Tornado 2, Microsoft MASM, Borland TASM, NASM/NASMIDE, HIWARE.*

CPLD/FPGA: VHDL, IP Core (PicoBlaze).
- *Modelsim, Xilinx ISE.*

Traitement du signal numérique: Filtres numériques et acquisition de données.
- *Matlab & Simulink.*

Analogique:

Electronique industrielle: Electrotechnique, automatique, électronique de puissance.
Electronique RF: Hautes fréquences, AM, FM, antennes, oscillateurs.
Traitement du signal analogique: Filtres analogiques, modulation.

Blaise LENGRAND
Page 2

EXPERIENCE

Développement d'un commutateur audio. *06/2006 – 09/2006*
Project54 – UNH, Durham, New Hampshire, USA

J'ai travaillé entant que stagiaire à l'Université du New Hampshire sur le Project54 (www.project54.unh.edu). Mon travail consistait à développer un logiciel capable de contrôler une carte son d'un PC afin de commuter n'importe quel flux audio provenant d'une ou de plusieurs entrées audio vers une ou plusieurs sorties de la carte. J'ai aussi développé un normaliser audio avec l'aide de Matlab.

Création d'un site web et de Frogans. *10/2005 - 12/2005*
FJPartners, Paris, Ile de France, FRANCE

Ce travail a été fait durant mes études par Internet pour une nouvelle société appelée FJPartners. Il consistait à développer de petites pages web pour une nouvelle technologie appelée "Frogans" (http://www.frogans.com). J'ai travaillé en collaboration avec un designer.

Développement d'algorithmes en C et en Caml. *10/2005 - 11/2005*
Marseilles, Provence-Alpes-Côte d'Azur, FRANCE

J'ai trouvé ce travail sur internet. Le but a été de développer 2 algorithmes pour un particulier. L'un en C consistant à implémenter un algorithme permettant de déterminer la coloration des arbres non orientés et un autre en Caml traitant de la "décision propositionnelle".

Traducteur d'un texte écrit en alphabet occidental vers l'alphabet Devanagari. *10/2004 - 11/2004*
Marseilles, Provence-Alpes-Côte d'Azur, FRANCE

Ce travail a été fait par Internet pour un particulier. C'est un programme développé en C qui, d'un texte écrit en alphabet occidental, crée une page web contenant la traduction en alphabet devanagari.

Création d'un site web. *7/2004 - 8/2004*
Kingston Ireland, Dublin, IRELAND

Mon travail consistait à développer des pages web en ASP avec Visual Basic .NET et une base SQL. J'ai appris à utiliser Microsoft Visio pour créer des diagrammes UML et ainsi générer les requêtes nécessaires à la création d'une base de données SQL. Le but des pages web que j'ai développées était de recenser le matériel informatique de l'entreprise.

ACTIVITES EXTRA-SCOLAIRES

11/2005 – 09/2006	Vainqueur du prix "Spécial Jeune" entant qu'inventeur et développeur d'un projet innovant sponsorisé par l'incubateur d'Angers Technopole (www.angerstechnopole.com) et la région du Maine et Loire. Ceci dans le but de créer une entreprise innovante. (http://www.angers.cci.fr/popups/popup_aa.html?id=566&type_ev=actualites).
2005 - 2006	Membre actif de la "Junior Entreprise" de mon école entant que développeur.
2004 - 2005	Membre actif du club robotique de mon école dans l'équipe de Recherche et Développement.
2004	Participant et finaliste du concours national de programmation « Prologin ». Le but était de développer le meilleur algorithme d'Intelligence Artificielle afin de contrôler un robot dans son environnement virtuel.

REALISATIONS

2005-2006
- Développement basé sur le noyau VXWorks tournant sur un Power PC du contrôleur d'un robot. Tout le code a été écrit en C avec Tornado 2 IDE. Les tests ont été effectués sur un simulateur tournant sous Linux.
- Programmation d'un DSP pour contrôler un moteur "brushless" en y incorporant un régulateur PID et un observateur d'état.
- Création de deux cartes à microcontrôleurs PIC communicant par liaison RF.
- Conception et création d'un amplificateur et d'un oscillateur HF pour être utilisé dans un démodulateur FM.

2004-2005
- Développement d'un jeu de « bataille navale » en Java. Ce jeu inclus une vision du plateau en 3 dimensions sans utiliser aucun composant Java déjà existant ainsi qu'une interface multi joueur basée sur les sockets Java.
- Développement d'un jeu de « Bomberman » intégralement en assembleur sur un microprocesseur Motorola 68K. Ce jeu a été développé pour être joué via un hyperterminal et inclus un générateur de carte pseudo aléatoire.
  J'ai aussi développé un jeu de « serpent » sur la même plateforme.
- Création d'un système électronique utilisant une PLL pour la restitution d'un son en utilisant uniquement des composants analogiques. J'ai aussi développé une PLL pour être utilisé sur cette carte entant qu'extension avec l'aide de Matlab.
- Développement d'un jeu de "nim" en langage C sous Linux en utilisant un réseau basé sur les sockets pour le mode multi joueur. Il inclus aussi un module de « chat » pour la communication instantanée avec les autres joueurs du réseau.

2003-2004
- Développement d'une application dans le but de générer des diagrammes de Vornoï en langage C. Ce code utilise la bibliothèque graphique SDL et est compilable aussi bien sous Linux que Windows.
- Développement d'une application appelé « SpriteCreator » fait pour ajuster la position de sprites pour être animés (http://www.ff6sol.com/).
- Création d'un système d'exploitation appelé « CryptOS », un noyau 32bits monolithique pour processeur i386+ écrit en assembleur Intel.
  Cet O.S. a un très simple contrôleur d'organisation de la mémoire (MM) et ne peu exécuter qu'un programme à la fois. Je l'ai réalisé dans le but de me familiariser avec la programmation bas niveau.

PASSE-TEMPS

Sports: Planche à voile (10 ans), badminton (4 ans), plongée (2 ans), voile, natation.
Loisirs: Peinture, calligraphie.

Introduction

In the context of my studies, I am currently carrying out my final training course with the Application Laboratory of the company Atmel Nantes S.A., under the responsibility of Mr. Cyrille Boulanger. The objective of this training course is to acquire a significant experience in the business world and to develop my skills to the maximum, relational as much as technical.

The branch of Atmel Nantes specializes in the design of integrated circuits of microcomputer type. I received as my mission to show the performance of 32-bit Atmel's microcontrollers (AVR32) by developing a digital signal processing library.

After a presentation of Atmel's company and more specifically, the Application Laboratory where my training course took place, this document presents in details the project upon which I worked on, various work accomplished, and finally a technical and human assessment.

Company presentation

1. Historical and geographical presentation

Atmel's company (Advanced technology for memory and logic) is relatively young. Until these last years, it never ceased growing. Atmel was founded in 1984 by an engineer, G. Perlegos, who employed 8700 employees all over the world. Specialized in the microcontroller, memories and ASICs domains, it was pointed out thanks to the design of a new memory: the flash. Nowadays, this memory is integrated in the microcontrollers and particularly in the AVR. So, Atmel is one of the leaders of the design and the integrated component in CMOS manufacture market.

The company is present in Europe, North America and Asia (**Figure 1**). The applicative fields of the manufactured component are various: automotive, space, multi-media, communication, military and safety. In the **figure 2**, we can see the places of manufacture and tests in the world.

Figure 1: ATMEL in the world: Headquarters of product design

Figure 2: ATMEL in the world: Headquarters of test, assemblage and manufacture

2. *The Business Units*

The organization of the research and the development is broken down by business units. They are organized according to the products which they design.
The different Business Units are:

- **Memory**: including the Flash, the Data Flash, EPROM
- **ASIC**: including CBIC, Smart Card and gate array products
- **RF**: including the RF and Automotive group of Heilbronn, the Analog and RF group of Duisburg.
- **Microcontroller:** including the microcontrollers of San José, the AVR group of Norway and the products group of Nantes.

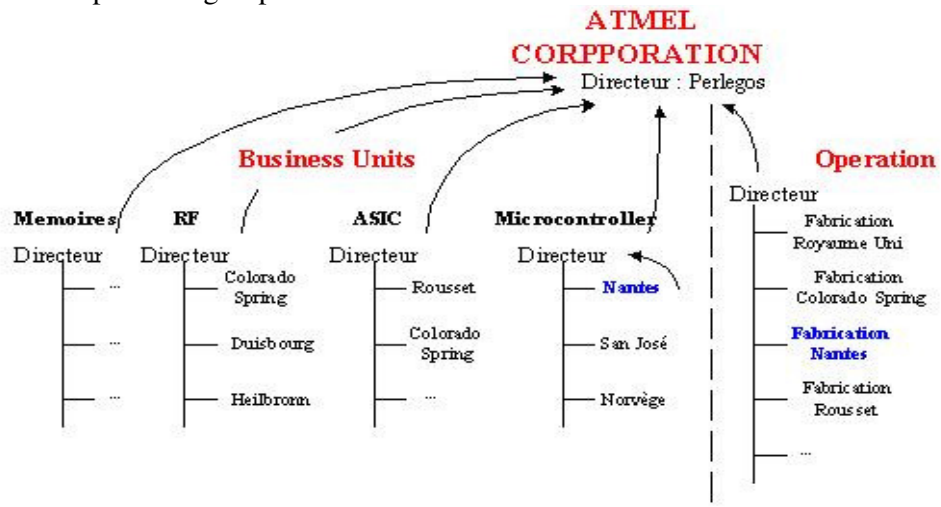


Figure 3: ATMEL Corporation and its BU

3. The manufacture

ATMEL designs and manufactures its products. The company has several workshops (**figure 3**). The technologies used for the manufacture are various. The ranges of process allow the optimization of the performances like the consumption of the products manufactured.
Here main technologies:
CMOS for the high performances digital applications and analog blocks,
CMOS with non-volatile memory for integrated memories like Flash or EEPROM,
BCDMOS et BCDMOS-SOI with high levels of tension, high temperatures and an immunity against the noises for the automotives and aerospace applications, Silicon Germanium (SiGe), Bipolar and BiCMOS for the radio and high frequencies interfaces.
Today, ATMEL makes more than 4 millions of components a day all over the world and more than 1 billion components each year.

4. Products and market

ATMEL is one of the largest producers of non-volatile memories like EEPROM and DATAFLASH in the world. Also, ATMEL designs and manufactures PLD, FPGA, ASICS, microcontrollers with 80c51 architecture (with integrated memory), microcontrollers with RISC (AVR) architecture and components for specifics applications.

Here the diagram of the split of the products' sales:

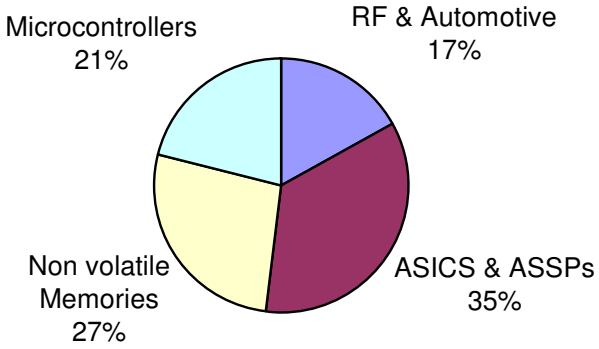


Figure 4: Revenue split by business unit

The components which are designed are used in various applications of the industry.
Here the diagram giving this split by application:

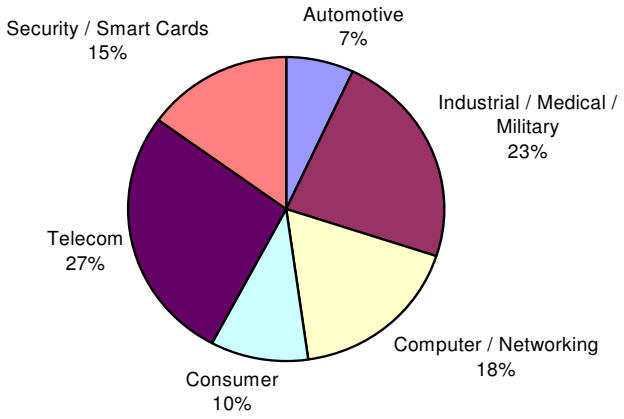


Figure 5: Revenue split by application

Each application is concerned with a certain number of products:

Applications	**Products**
Computer / Networking:	
Wireless LANs	WiFi Media Access Controller
PC and PDA	USB controllers
Exchangeable Data Storage	Storage Network Interface
…	…
→ *System on chip products with embedded RISC microcontrollers*	
Consumer:	
DVD and MP3 Players	DVD Chipset and Laser Controller Ics
Music Synthesis / Karaoke	MP3 Decoder
Electronic Games	ARM-based 32 bit Microcontrollers
Digital Cameras	AVR 8 bits Microcontrollers
….	…
→ *High volume, High integration, low cost, low power consumption, ASSP and ASIC*	
Telecommunications:	
GSM / GPRS / UMTS Mobile Phones	GSM / GPRS Baseband Processor
	RF Transceivers
Secure Mobile Radios	Miniature Digital Camera Modules
Global Positioning System (GPS)	Power Management ICs
….	…
→ *Highly integrated analog digital and RF Systems On Chips*	
Security / Smart Cards:	
Banking and Finance	Secure Microcontrollers
PC security	Secure memories
Asset Management	Finger Chips
Manufacturing / Logistics	Trusted Platform Module
…	…
→ *Atmel is a market leader in secure Rash based microcontrollers*	
Automotive:	
Vehicle Body Electronics	Vehicle Network / Multiplexing (CAN, VAN, USB, …)
Vehicle Access	
Power Train Management	GPS Receiver and Baseband Processor Chipset
…	…

→ *Market leading products conforming to stringent automotive industry standards*	
Industrial / Medical / Military: Industrial: Digital Cameras, Transportation…. Medical: Blood Glucose Metering… Defence / Space: Satellites / Radars… Avionic: Flight Control…	 CCD Image Sensor High Reliability Microcontrollers Embedded Wireless Transceivers …
→ *Fully Qualified to military standards*	

Here is the diagram giving the sales split of ATMEL around the world. We can see that the company sells especially in Asia and in Europe and a little less in USA and Canada.

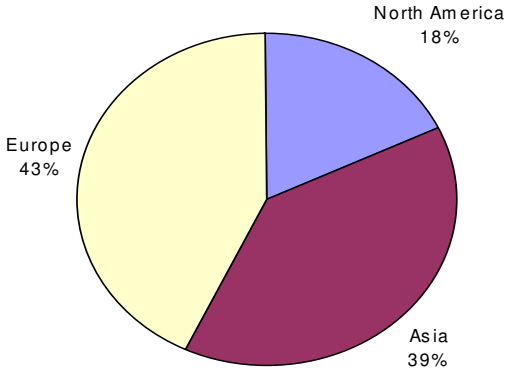


Figure 6: Revenue split by region

Here is the diagram giving the evaluation of the company income with respect to the total semi conductor market income:

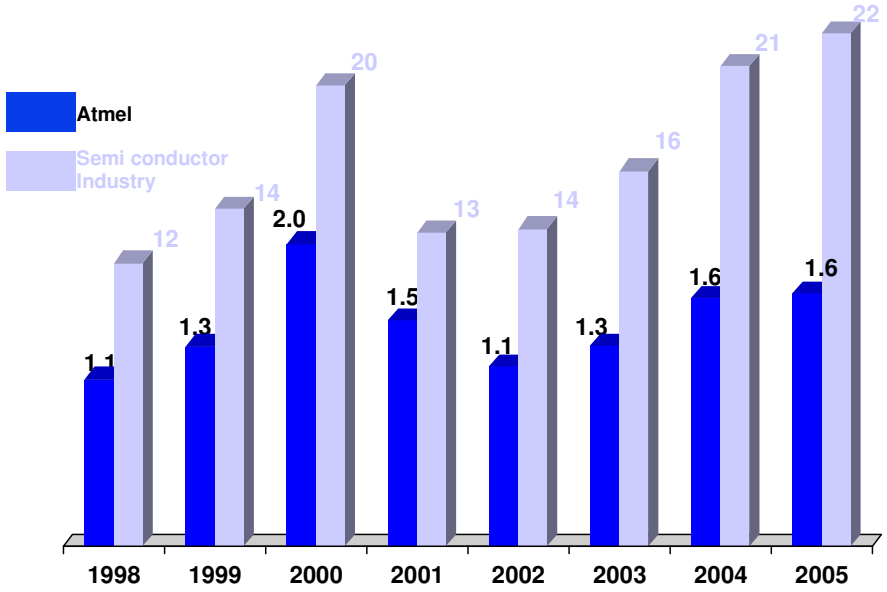


Figure 7: ATMEL income ($billion)

5. The site of Nantes

The site of Nantes was built in 1979 by MATRA HARRIS (MHS) semi-conductor at 8 km of the center of the city. Then, the company has changed his name to become Temic and later Atmel Wireless & Micro-controllers. The site of Nantes is composed of 287 persons and counts of about 78% are engineers.
The manufacture made components. ATMEL Nantes includes 3000m² of clean rooms classified 1 and 10.

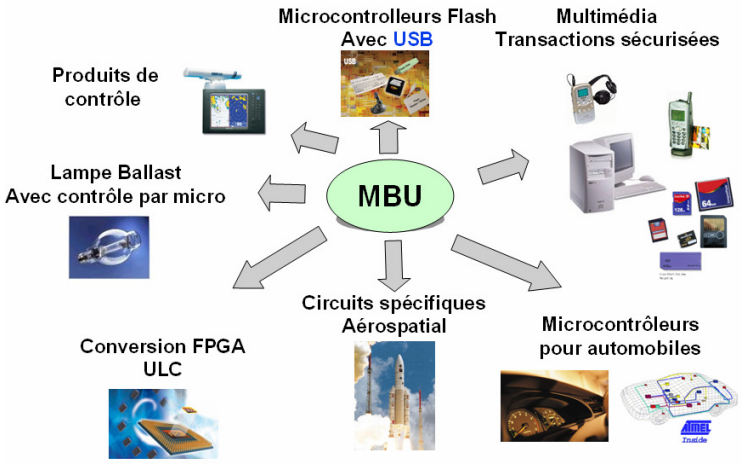


Figure 8: ATMEL NANTES Products

The average age is 41 years and 24% of the total staff is women.

6. Application Laboratory

The application laboratory, commonly called "labo d'appli", is the department where I am currently working.
The Application lab is one of the many services of the design part in Atmel Nantes, and is a party of the "Micro controllers Business Unit". This department is directed by Mr. Berthy who manages the engineers and assigns them to the different projects in progress.

Roles of applications engineers:

- Analyze the customer requirements precisely and to take part in the product's specification.
- Write datasheet's products.
- Write the software (boot loaders) to program the micro controllers in ISP (In System Programming) mode.
- Write the peripherals drivers for the micro controllers (USB, CAN, MP3...).
- Specify, design and build the demonstrations boards to present the product to the customers.
- Write the demonstration software.
- Provide a hotline support for customers.
- Maintain contacts with the subcontractors (like PCB subcontracting).
- Test and validate the products with the tools that a client could have.

AVR32UC3 Software Framework

First of all, it is important to introduce the notion of "Software Framework" which is the cause for much of the work done by the engineers of the Application Laboratory.
The AVR32UC3 Software Framework consists of AVR32UC3 microcontroller drivers, software services & libraries, and demonstration applications.
Each software module is provided with a full source code, example of usage, rich html documentation and ready-to-use projects for the IAR EWAVR32 and GNU GCC compilers. It also comes with a pre-built workspace for AVR®32Studio (an Eclipse based IDE developed by Atmel).

The AVR32UC3 Software Framework is made of the following sub-packages:

- **The drivers sub-package** - It contains software drivers, board-specific (EVK) C/C++ files with defines, macros and functions and other useful files used by all other directories of the library.
- **The Services sub-package** - It contains application-oriented pieces of software that are not specific to boards nor chips and software interfaces to interact with the component.
- **The Applications sub-package** - It contains hefty examples of applications using the Services and Drivers sub-packages.

Here is a simple representation of the dependencies between the three UC3 software sub-packages:

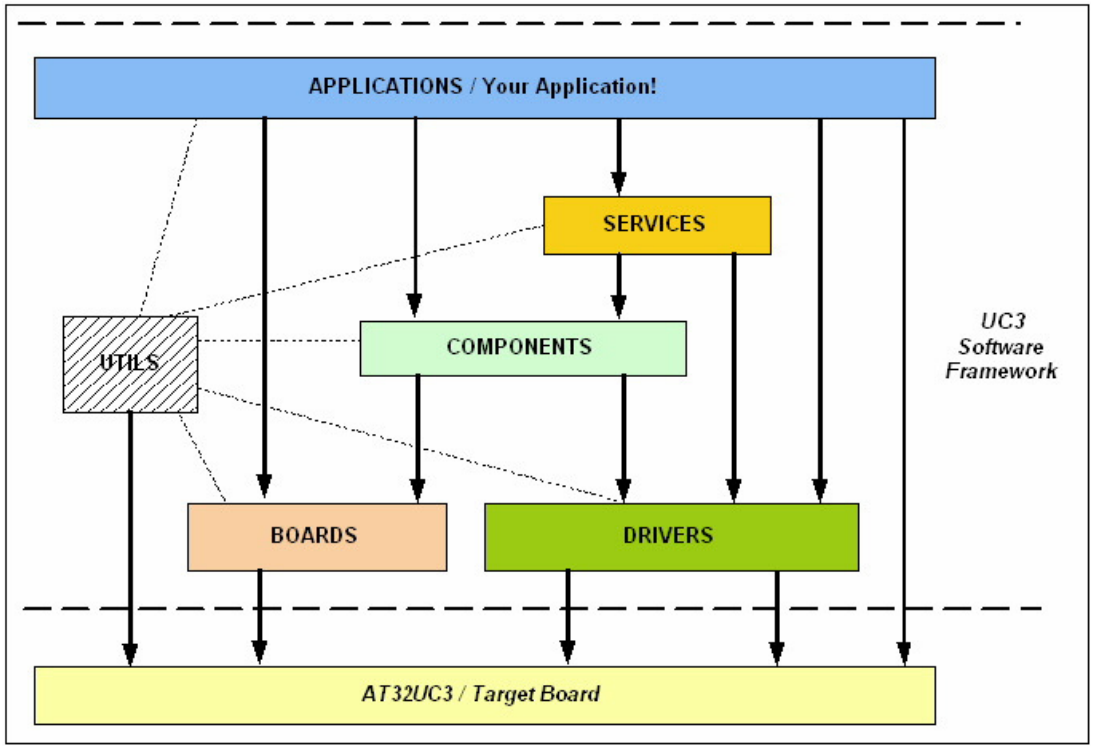


Figure 9: UC3 software packages dependencies

Requirements

The purpose of this mission is to write a software library that provides optimized digital signal processing functions for the AVR32UC3 microcontroller.

This project mainly consists of:

- Identifying and specifying the main DSP algorithms to include in the library.
- The proficiency of using assembly language and DSP instructions.
- Conceiving, developing and validating an optimized DSP library that provides the main filters and algorithms used nowadays in DSP applications.
- Profiling functions using benchmarks and comparison with the concurrency.

This library will be available for customers as a free DSP software solution downloadable on the Atmel's website on the package AVR32UC3 Software Framework.

AVR32UC3 product specifications

1. Introduction

ATMEL has created the first processor architected specifically for 21st century applications that require both performance and low power consumption. The AVR32 32-bit RISC processor core is designed to do more processing per clock cycle so the same throughput can be achieved at a lower clock frequency with substantially less power consumption.

2. Features overview

The Atmel AVR32UC3 product family is built on the new AVR32 UC core optimized for highly integrated embedded applications requiring microcontrollers with on-chip Flash program memory, high computation throughput, real-time behavior and low power consumption.

- High computation throughputs: The AVR32 UC core is single cycle operation and has a tightly coupled SRAM.
- Deterministic & real-time control: The Atomic read-modify-write (bit banding) and fast interrupt response provide fine event control.
- Low power: AVR32 UC core works smart. It requires lower bus activity and lower operating frequency than other architectures.
- Low system cost: The Instruction set gives high code density resulting in smaller memory usage.
- High reliability: The AVR32 UC safely boots from the RC-oscillator. The NMI ensures critical event processing and the MPU secures memory accesses.
- Easy to use: High performance and code density allows software designers to use high level programming methodologies without execution speed or cost penalty.

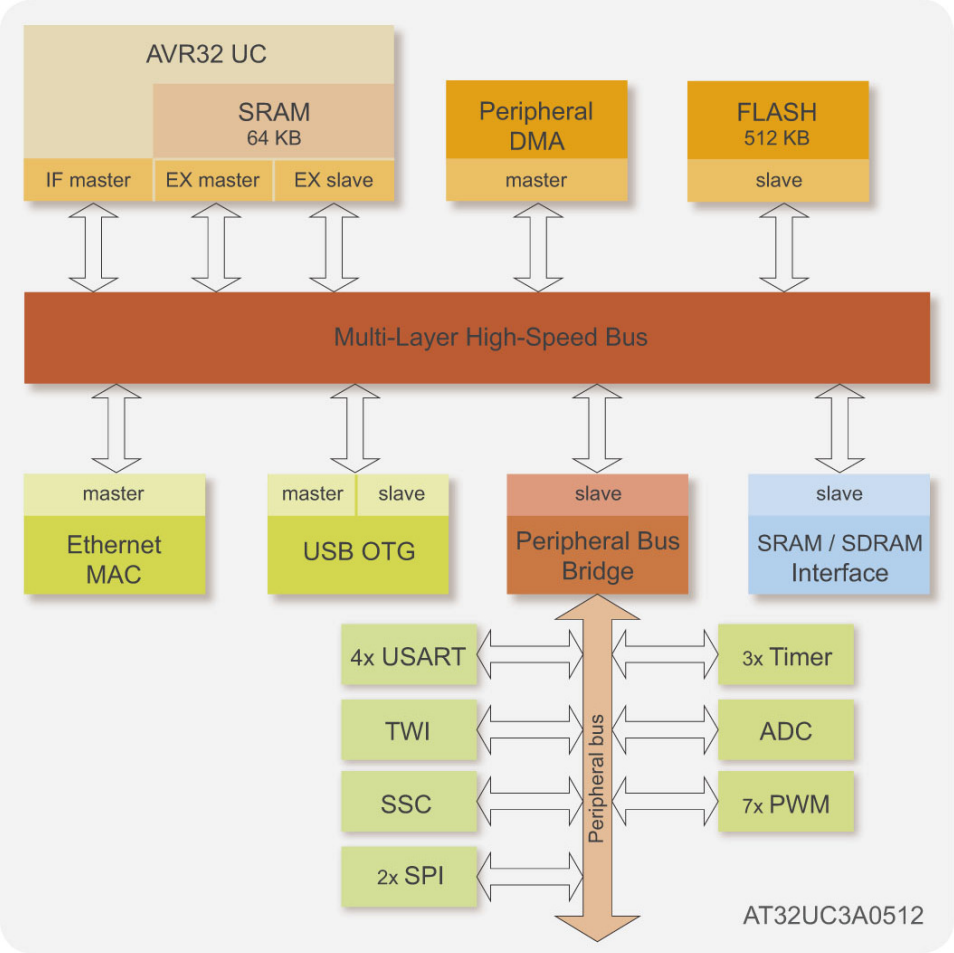


Figure 10: AVR32UC3 A Series block diagram

Features:

- 32- to 64 Kbytes SRAM
- 128- to 512 Kbytes Flash
- 10/100 Ethernet MAC
- Full-Speed (12 Mbps) USB 2.0 with On-The-Go (OTG)
- USB Boot loader
- SRAM/SDRAM external bus interface.
- 13 timers (Timer/counter, PWM...)
- 4 USARTS
- 2 SPI + 1 SSC
- 1 TWI (I2C)
- 8-channel 10-bit ADC
- 1 RC-Oscillator
- 32 kHz oscillator
- 2 oscillators & 2 PLLs
- Power Manager (POR, BOD)
- 100-144-pin QFP package

3. Core overview

- **It is the first Core to integrate SRAM in the pipeline** - The AVR32UC core is the first core in the industry to integrate single-cycle read/write SRAM with a direct interface to the CPU that bypasses the system bus to achieve faster execution, cycle determinism and lower power consumption.

- **3-Stage Single-cycle Pipeline -** The core is based on a simple 3-stages single cycle pipeline that includes a "prefetch", a "decode" and an "execution" unit.

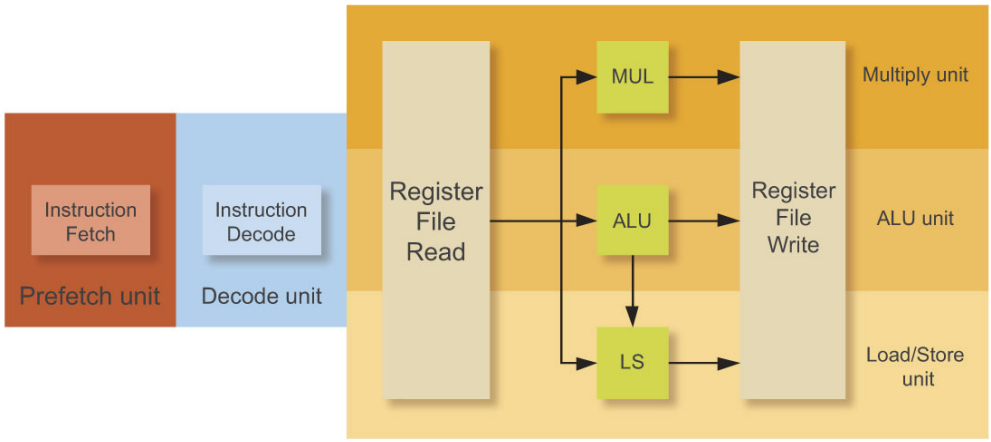


Figure 11: Pipeline stages of the AVR32UC core

- **Instruction Set Architecture with freely intermixable 16/32-bit instructions -** The AVR32 UC core shares the same instruction set architecture (ISA) as its AVR32 AP parent, with over 220 instructions available as 16-bit compact and 32-bit extended instructions. The AVR32 ISA is designed to minimize data transactions between the core and memories, saving both power and clock cycles.

- **DSP Instructions -** The AVR32 UC core multiply-accumulate unit executes, in a single cycle, multiple of multiply and multiply-and-accumulate instructions on standard and fractional numbers, with and without saturation and rounding. Multiply or MAC results can be 32-, 48- or 64-bit wide; 48- and 64-bit results are placed in two registers. DSP instructions also include many add and subtract instructions as well as data formatting instructions such as data shift with saturation and rounding.

- **Fast Event Handling** - The AVR32UC core event handling system support events like non-maskable interrupt (NMI), exceptions (illegal opcode, bus error), and four interrupt priority levels.

- **20% Better Code Density than ARM7 (Thumb) or Cortex M3 (Thumb2)** - AVR32 UC code is consistently 5% to 20% smaller than code compiled for the ARM Thumb ® instruction set.

Training period

First of all, in order to familiarize myself with the tool chain used by the "Application Laboratory", Cyrille Boulanger advised me to develop a driver for the AVR32UC3.
He chose a simple driver not yet implemented: the Real Time Counter.

1. Tools

To do so, I had at my disposition an evaluation board, the EVK1100 which is an evaluation kit and a development system for the AVR32 AT32UC3A microcontroller.
It is equipped with a rich set of peripherals, memory, and makes it easy to try the full potential of the AVR32 devices.

- Supports the AT32UC3A
- Ethernet port
- Sensors: Light, Temperature, Potentiometer
- 4x20 Blue LCD (PWM Adjustable backlight)
- Connectors for JTAG, Nexus, USART, USB 2.0, TWI, SPI
- SD and MMC Card Reader

This board was mounted with an AVR32UC3A0512 microcontroller which specificity is that it contains 512Kbytes of FLASH memory.

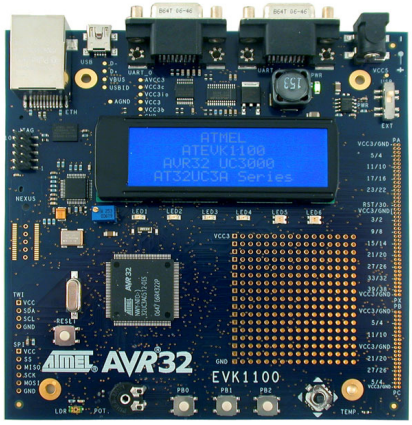


Figure 12: EVK1100 evaluation kit

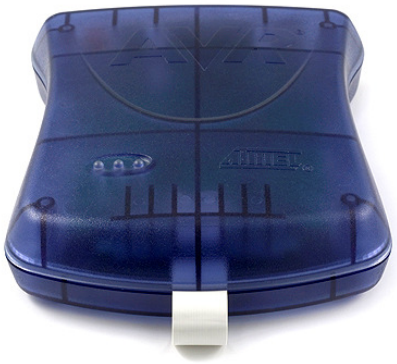
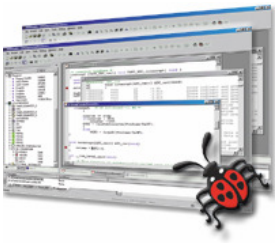


Figure 13: JTAG ICE mkII

In order to program the AVR32UC30512 microcontroller found on this board, I had in my disposition a JTAG ICE mkII which is a low cost In-Circuit Emulator supporting all AVR and AVR32 with debug WIRE or JTAG interface.



AVR®32Studio

AVR32-GNU
tool chain

I also had in my disposition a PC running under Windows XP Pro, and Cygwin to emulate an UNIX platform. The software tool chain development for AVR32 products is made of many identities. All of the software development should be compatible with IAR and the GNU tool chain. Therefore I had to become familiar with those tools. We made several projects using GCC and GDB at school and the AVR32-GNU tools are pretty similar. The innovation for me was IAR. I have heard good things about this compiler before coming here, things that revealed to be true once I started using it. It is really simple to use and the IDE they provide is clear and easy to access. It also provides a debugger tool that is well integrated in the interface. The only disadvantage is that it is a shareware which needs a non-free license to run. The Application Laboratory of Atmel owns ten licenses but only three are accessible because of a bug that IAR is currently resolving in their license server.

To develop in an incremental and controlled fashion, we used a revision control system called SVN. We used more specifically the interface called TortoiseSVN, available for Windows and totally integrated in the Microsoft file explorer which procures a great simplicity of utilization.

2. RTC driver

The Real Time Counter (RTC) enables periodic interrupts at long intervals, or accurate measurement of real-time sequences. The RTC is fed from a 16-bit prescaler, which is clocked from the RC oscillator or the 32 KHz oscillator. Any tapping of the prescaler can be selected as clock source for the RTC, enabling both high resolution and long timeouts. The prescaler cannot be written directly, but can be cleared by the user. The RTC can generate an interrupt when the counter wraps around the value stored in the top register, producing accurate periodic interrupts.

The following figure gives an overview of the RTC.

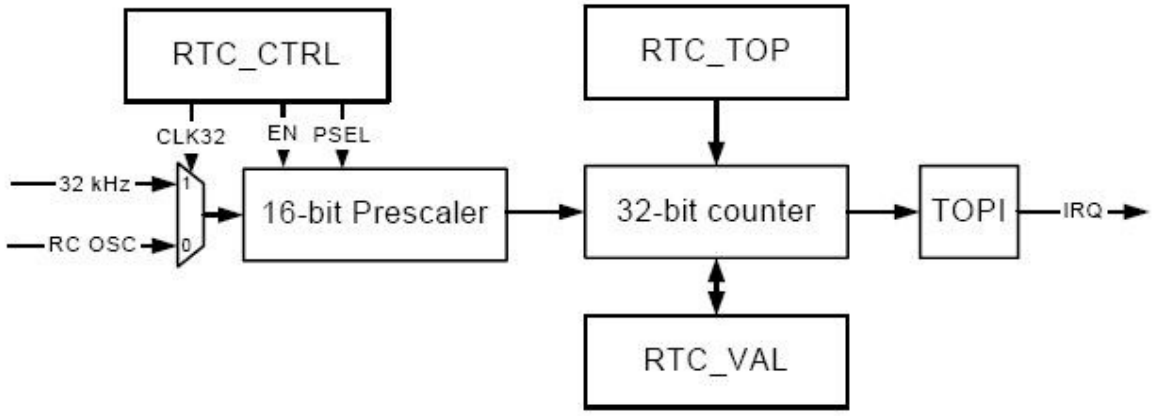


Figure 14: Overview of the Real Time Counter (extracted from datasheet)

To develop this driver I used the AVR32UC3A datasheet frequently which was not released publicly at this time and so all the information was not verified.
First, I used IAR to develop and debug the driver and then I compiled it with GCC compiler to make sure the code was compatible.
Then I wrote an example for how to use this driver.
Finally, I documented all the code in Doxygen format and made a little documentation for the example I created.

Here is a Gantt diagram that shows the training period:

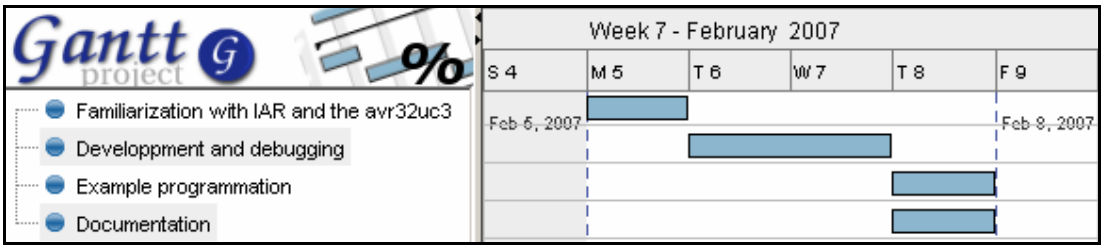


Figure 15: Gantt diagram of the training period

3. *Bugzilla*

During the development of this driver I noticed some anomalies on the datasheet and in the header files, here is the list I established:

- The datasheet says you can access the 16 bits of the “Prescale Select” value (psel) on the RTC control register (CTRL reg.). But if you set this value to be greater than 15 (more than 4bits), the psel value resets to 0.
- pm_enable_osc32_crystal() : wrong name (no enable performed).
- The constants AVR32_PM_OSCCTRL1_MODE_CRYSTAL, AVR32_PM_OSCCTRL0_MODE_CRYSTAL and AVR32_PM_CRYSTAL are defined in the file « pm_200.h » with the value 7. According to the datasheet this value should be 1.

- In the function pm_enable_osc32_crystal() from the driver PM, the field "mode" of the OSCCTRL32 register, receives the value AVR32_PM_CRYSTAL (=7). And this constant is not defined according to the datasheet in the file "pm_200.h". This value should be 1 and it appears that this function doesn't work properly with the value AVR32_PM_CRYSTAL.

→Those two last bugs have been corrected by modifying the associated header files.
→The "pm_enable_osc32_crystal" naming rules were not a bug because "enable" means in fact that the 32 kHz oscillator crystal is "connected to the RTC" and ready to be used.
→And finally, the bug on the datasheet has been reported on Bugzilla (it is a free solution for bug-tracking used by Atmel to report every bug in their tools or their products).

Bugzilla report:

The PSEL bit-field of the RTC CTRL register is 4 bits wide, not 16.

DSP library specifications

1. Functions

At the startup of the project, I had to begin all by myself. Of course I had to follow the requirements but nothing was specified for the functions to implement and neither for the architecture of the library. Nothing significant was done by Atmel in the Digital Signal Processing domain. The only person who could help me to specify this library was the marketer of the product, who, by the actual demands on the market could advise me and could require specific functions to be implemented.
Therefore, I talked to Regis Latawiec and established a list of several important functions I had to specify. He also wanted a library with not only functions, but also real applications and mainly in the telephony domain.
The other path of my research was on the work already done by the concurrency. I asked Regis about that and did my own research on the web too. I found several free and not free DSP libraries and based my pursuits on the dsPIC and the STR91x DSP libraries, because these two microcontrollers are the main concurrents of the AVR32UC3.
Finally, I have tried to copy as much as possible, the functions prototypes implemented in Matlab and Scilab, in order to provide a library that can be easily compatible with a Matlab or a Scilab script.

I designed the library in two sub-libraries: one for basic functions (called basic library), such as filters and transforms, and the other, based on the first one, containing the applications (called advanced library).
The basic library is also divided into 7 modules. Here is a list of the functions specified and sorted by modules:

→ **Operators**
- Cosine
- Sine
- Arc cosine
- Arc sine
- Square root
- Power
- Exponential
- Natural logarithm

→ **Vectors**
- Addition
- Subtraction
- Multiplication
- Dot Power
- Division
- Dot multiplication
- Dot division
- Convolution
- Minimum
- Maximum
- Normalize
- Cross correlation
- Zero padding
- Copy
- Negate

→ **Transforms**
- Fast Fourier Transform (FFT)
- Complex Fast Fourier Transform (complex FFT)
- Inverse Fast Fourier Transform (IFFT)
- Discrete Cosine Transform (DCT)
- Inverse Discrete Cosine Transform (IDCT)

→ **Filtering**
- Finite Impulse Response (FIR)
- Infinite Impulse Response (IIR)
- Least Mean Square (LMS)

→ **Windowing**
- Rectangular
- Triangular
- Blackman
- Hamming
- Gauss
- Hanning
- Kaiser
- Bartlett

→ **Signal generation**
- Sine
- Cosine
- Square
- Triangular
- Ramp
- Echelon
- Noise
- White noise

- Rectangle
- Dirac comb
- Dirac
- Pink noise

→ **Debug**

- Print real
- Print complex
- Print real vector
- Print complex vector

This is a list of the applications implemented on the advanced library:

→ **Telephony**

- Echo cancellation
- Calling Number Identification (CNID)
- Automatic Gain Control (AGC)
- Dual Tone Multi Frequency detection (DTMF)

→ **Soft Modem**

- V90 – V91

→ **Compression**

- Adaptive Differential Pulse Code Modulation (ADPCM)

→**Regulator**

- Proportional Integral Derivative (PID)

In order to use all the capacity of the microcontroller, Regis, Cyrille and I decided to design for each function a 16-bit and a 32-bit version; therefore, all functions had to be coded two times.

2. *Types*

The AVR32UC3 does not have a floating point unit. Thus, I had two choices: either I emulated a floating point unit using the compiler intrinsic functions; or I use a fixed point format for the decimal types. The choice was easy: software floating point unit emulation would be too slow to perform, that's why I used a fixed point type.
Each function is designed to work with 16 and 32 bits signed values. The 16 bits type is used when you favor speed instead of accuracy or else you can use the 32 bits type.
All types include the fixed point notion which default values allowed are in the range [-1; +1]. To implement this notion, I chose the Q-Format: Q1.15 for the 16 bits type and Q1.31 for the 32 bits type, this to be the most compatible with the existing DSP libraries.

Here is a description of the Q-Format:

Q-format: Qa.b

a *is the number of bits used to defined the integer value.*
b *is to define the number of bits used after the radix point.*

Here is the formula linking a Q-format number (x) to a decimal number (d):

$$d = \frac{1}{2^b}.\left[-2^{N-1}x_{N-1} + \sum_{n=0}^{N-2} 2^n x_n\right]$$

x_n : *The value of the bit number n of the Q-formatted number x.*

N : *The number of bits of the Q-formatted number x.*

The resolution of a Q-formatted number is calculated as follow:

$$resolution = \frac{1}{2^b}$$

The range of a Q-formatted number x defines the maximum and minimum numbers which frame x.

Here is a formula to calculate those values:

$$-2^{N-b-1} \leq range \leq 2^{N-b-1} - \frac{1}{2^b}$$

N : *The number of bits of the Q-formatted number x.*

For the Q1.15 type:

Resolution: *0.00003*
Range: *-1.* $\leq range \leq$ *0.99997*

For the Q1.31 type:

Resolution: *0.00000000047*
Range: *-1.* $\leq range \leq$ *0.99999999953*

A complex type is a packed structure of two elements, the first one containing the real data and the second one the imaginary.
A vector is an aligned packed table of a basic type.

3. Function's naming convention

A function must follow this naming rule: dspX_y_z(…)
Where:
X is the type used (16 or 32)
y is category of the library (op, vect, trans, …)
z is the name of the function (cos, mul, …)
Each function name must have a maximal length of 32 characters to be ANSI compatible.

Example:
dsp16_vect_mul is a function part of the DSP library. It works with 16 bits values and permits to multiply vectors.

4. Compatibility

The whole Library has to be "IAR" and "GCC" compatible in order to fit in the AVR32UC3 Software Framework.
The optimized functions will not be written in C ANSI because they will contain assembly code in order to use the DSP instructions set provided by the AVR32UC3. Thus, to maximize the compatibility with others compilers or targets, all the functions will have a generic version written in C ANSI languages.
Therefore, at the maximum, a function will be coded in four different versions: a 16-bit and a 32-bit version for generic and the same for optimized functions.

5. Compilation options

When you compile the library, you can specify different options that permit you to optimize the algorithm according to your needs. Four different options are available:

- **DSP_OPTI_SPEED**: this option will speed optimize the algorithm used in the functions.
- **DSP_OPTI_SIZE**: this option will minimize the size of the algorithms and their resources.
- **DSP_OPTI_ACCURACY**: this option will increase the precision of the data computed by the algorithms.
- **DSP_OPTI_ACC_AND_SIZE**: this is the combination of size and accuracy optimization options.

By default, if no option is specified, the algorithms will be optimized to run as fast as possible.

Project management

Once I finished a first version of the specifications I decomposed the project into sub tasks in order to estimate the time needed more precisely. The next diagram summarizes the project steps and tasks in details; the task order is mainly sequential.

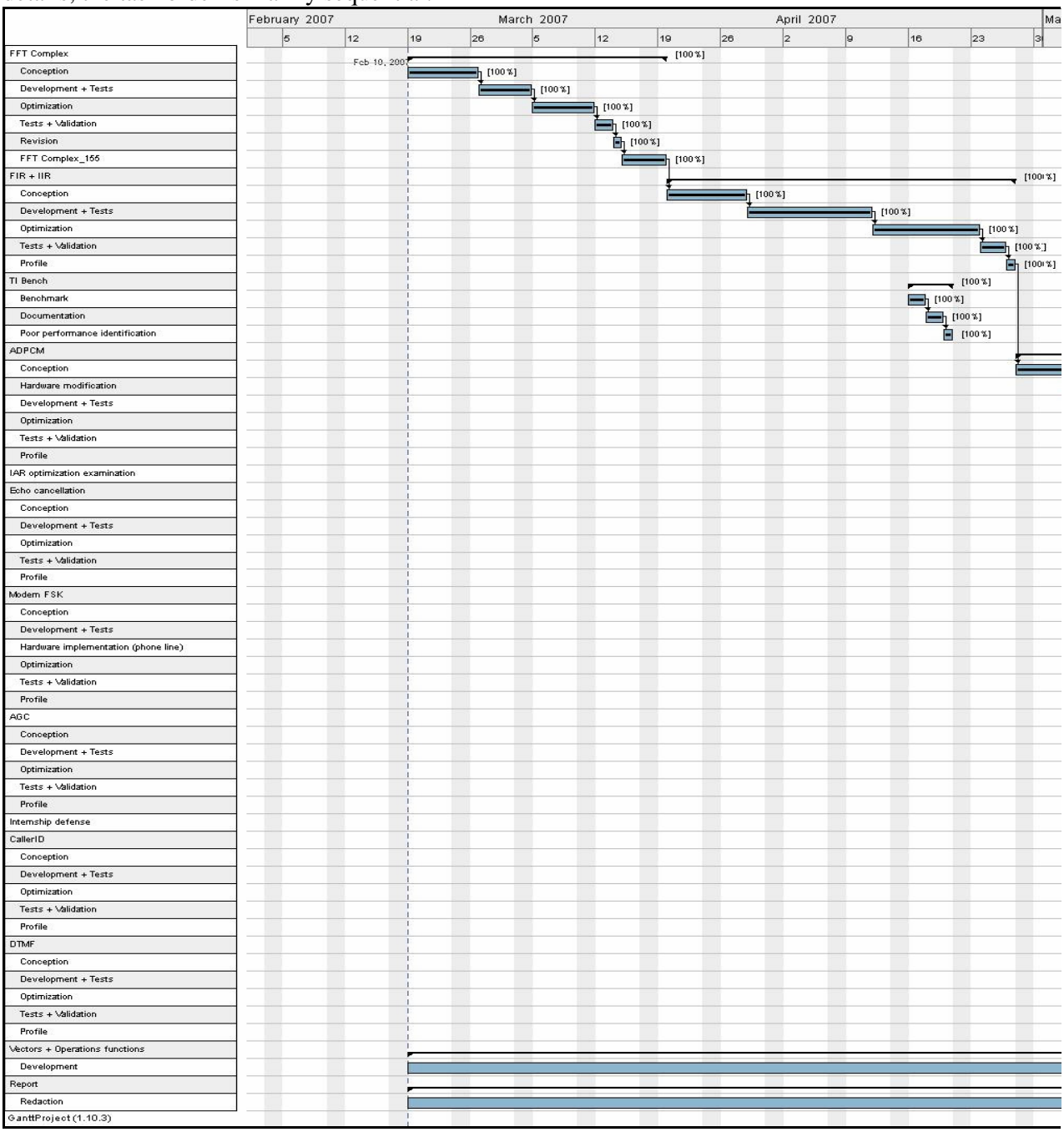


Figure 16: Gantt diagram (part 1)

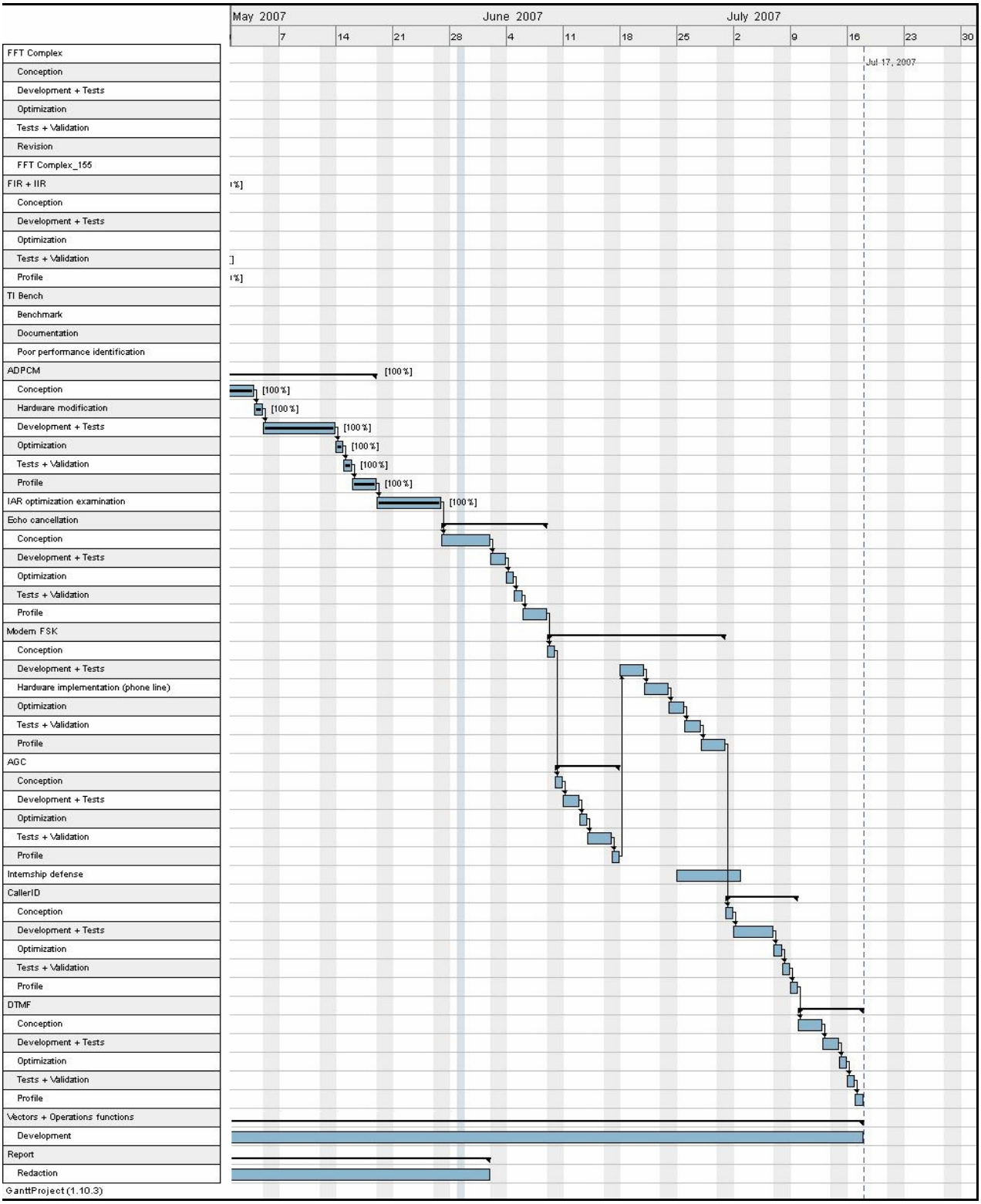


Figure 17: Gantt diagram (part 2)

Note: this diagram has been updated several times due to activity switching to do benchmarks or reports. Here is a table showing more precisely the actual schedule and including those unforeseen events.

Week number	Activity	Duration
6	Familiarization with IAR avr32 and the avr32uc3 microcontroller	1 day
	Development of the Real Time Counter driver and debugging	2 days ½
	Documenting and programming the example program for the RTC	1 day
	Starting the specifications of the DSP Library	½ day
7	Seeking information about DSP libraries	1 day
	Writing specifications and thinking about the best way to define types	4 days
8	Finalizing Requirement documentation	1 day
	Seeking documentation and codes about FFT algorithms	1 day
	Conception of FFT DIT (Decimate In Time) algorithms (2-radix, 4-radix and split-radix) with Scilab • DIT means that you modify first the order of the input vector and then, after you run the algorithm, the vector is all well sorted (in opposition to the DIF method). This must be the fastest way to process a FFT in our case, because it will avoid doing one loop to browse the vector. • Problem finding information on split-radix DIT algorithm and no code founded but I finally understood it. It is complex but faster than the 2-radix and the 4-radix algorithms. (STMicroelectronic provides a 4-radix algorithm but Microchip a split-radix)	3 days
	Committing my RTC driver	1 hour
9	Finalizing FFT DIT split-radix conception	1 day
	Programming FFT algorithm in C language and testing	3 days
	Starting the optimization	1 day
10	Optimizing the FFT DIT split-radix algorithm in assembly language	3 days
	Noticing bad performances because of the algorithm. It is more adapted for small microcontrollers (less multiplications)	½ day
	Starting a new algorithm implementation and a new data storage method	1 day ½
11	Specifying developments priority with Regis	2 hours
	Optimizing the FFT DIT 4-radix algorithm in C-language and testing	1 day
	Optimizing the FFT DIT 4-radix algorithm in assembly-language	2 days ½
	Testing the algorithm	1 day
	Profiling the functions	½ day
12	Profiling the assembly function and the C generic function	1 day ½
	Converting the code to be IAR compatible	½ day
	Making doxygen documentation of the code	1 day
	Making a script to automatically benchmark functions	1 day
	Researching information about CallerId, ADPCM, Echo cancellation, modem FSK and agreed with Regis about the planning	1 day
13	Enhancement of the complex FFT speed	1 day ½
	Profiling the assembly function and the C generic function	½ day
	Conception of the FIR algorithm	1 day
	Conception of the Sinus and cosinus operators for fixed point data	1 day
	Implementation of the Sinus and cosinus operators	1 day
14	Implementation of the cosinusoidal and sinusoidal generators	1 day
	Conception of the FIR algorithm	½ day

	Implementation of the FIR algorithm	1 day
	Optimization of the FIR algorithm	1 day
	Implementation of two functions for the FIR algorithm, one optimized for symmetric coefficients and one all kinds	½ day
15	Conception of the convolution algorithm	½ day
	Implementation of the convolution function	1 day
	Benchmarking the FIR and convolution functions	1 day
	Documenting FIR and convolution functions	1 day ½
16	Benchmarking the avr32uc3 with the TI BENCH	3 days
	Identifying the problem of poor performances of the uc3 and establishing a new bugzilla	1 day
	Cleaning code and adding new examples	1 day
17	Conception of the IIR with scilab	1 day
	Implementation in C language	1 day
	Optimization for the 16-bits version	1 day ½
	Optimization for the 32-bits version	½ day
	Documentation of the IIR functions	1 day
18	Documentation of the full DSP library, creation of readme.html	2 days
	Seeking information about the ADPCM algorithm (asking François Fosse and Anthony Rouaux) and taking a decision on the IMA/DVI algorithm	1 day
	Starting implementation of this algorithm	1 day
19	Implementation of a streaming IMA/DVI ADPCM sound player using the USART on the EVK1100 development board	2 days
	Writing a little report for my school	1 day + …
20	Documenting the ADPCM streaming application	1 day
	Finalizing the ADPCM functions	1 day
	Looking at IAR bad optimizations compared to GCC	1 day
	Starting to establish a CPU charge report for the ADPCM streaming example	½ day
	Richard Perdriau's visit	½ day
21	Tracking IAR bad optimizations	3 days
	Seeking echo cancellation documentation	½ day
	LMS filter conception and development	1 day ½

Note: every week, I wrote a report to inform Cyrille Boulanger about my advancements.

DSP library development

1. Library architecture

The library is made up of different subdirectories. Here is the architecture at the root of the library:

***Note**: The directory that appears in bold on the following figures will be released in the Software Framework. The others will be only used for the internal development at Atmel.*

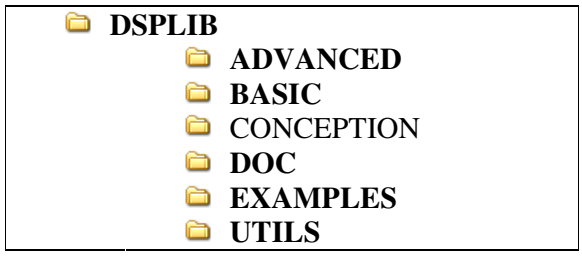


Figure 18: DSP library root directory architecture

The **ADVANCED** directory includes all the applications available in the DSP advanced library.
The **BASIC** directory includes all the functions provide in the DSP basic library.
The **CONCEPTION** directory regroups all the documents I have created while I designed the algorithms.
The **DOC** directory includes the main documentations of the whole library.
The **EXAMPLES** directory regroups a lot of examples showing a way to use the functions provided by both advanced and basic libraries.
The **UTILS** directory regroups useful tools and scripts for the DSP library.

The basic library counts different modules that can be seen in the directory architecture of the library:

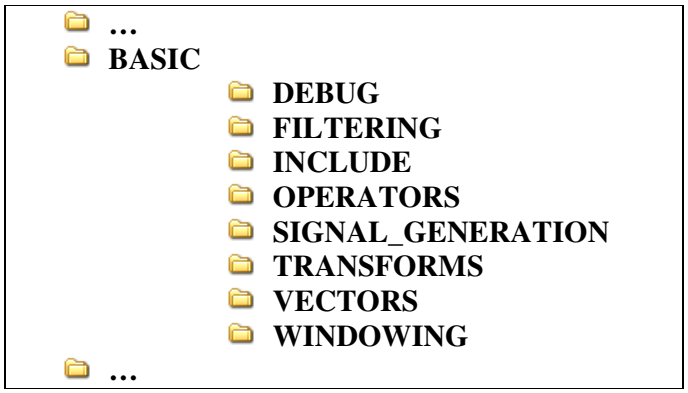


Figure 19: DSP library BASIC directory architecture

The **DEBUG**, **FILTERING**, **OPERATORS**, **SIGNAL_GENERATION**, **TRANSFORMS**, **VECTORS** and **WINDOWING** make up the 7 modules available in the basic library. In each of these directories, the C and assembly files are sorted and used to code the functions implemented in the corresponding modules.

The **INCLUDE** directory regroups all the main header files of the basic library:

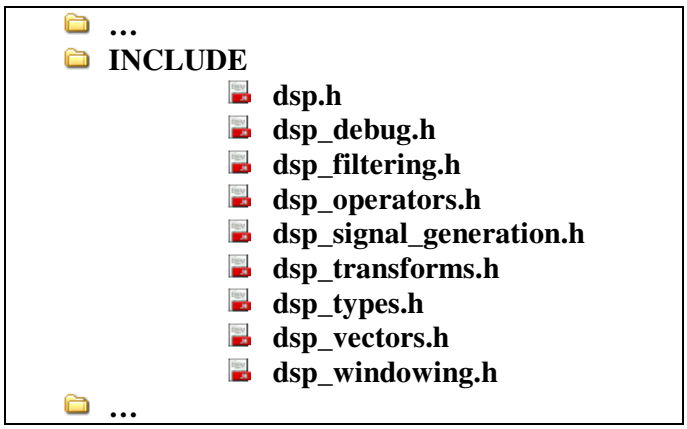


Figure 20: DSP library BASIC/INCLUDE directory architecture

In this directory appear only header files. The dsp_XX.h files, where XX corresponds to the name of a module, including the declarations of the functions implemented in their respective module. The **dsp_types.h** header file is the file that contains the declarations of the basic types and useful macros used in the DSP library. The file **dsp.h** is the only file the user has to include in his project in order to benefit from all of the DSP basic library functions.

The **ADVANCED** directory for the DSP advanced applications is based on the same architecture for the exception that the modules are in fact the applications. Example: instead of having a subdirectory called **VECTORS** like in the DSP basic library and a **dsp_vectors.h** file in the **INCLUDE** directory (where "vectors" is a module), you will have an **ADPCM** directory and an **adpcm.h** file, where "adpcm" is an application, but not a module.

The **EXAMPLES** directory is composed of many didactic examples for the whole library. For each example a directory is created whose name is the name of the example. Each example has the following directory architecture:

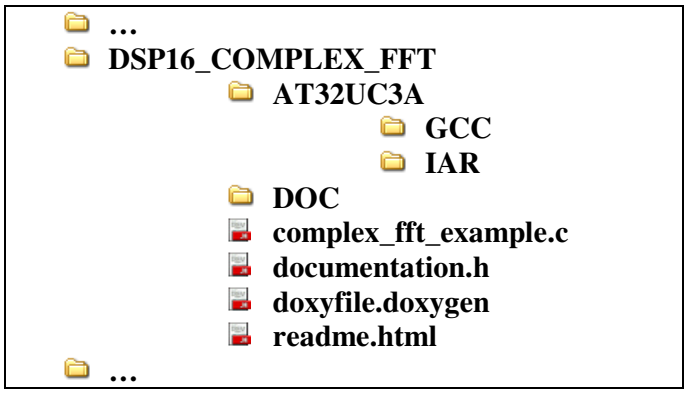


Figure 21: Example directory architecture

It is composed of two directory types. The first one, here named **AT32UC3A**, includes all building files relative to a specific architecture. For example, in this case, the complex FFT has an example with building files only available for an AT32UC3A target (which is the AVR32UC3 A series microcontroller). The second type of directory is the **DOC** directory which contains the Doxygen documentation.

This documentation is generated thanks to the file **doxyfile.doxygen** which contains all of the configuration information required by Doxygen to generate the documentation. This file defines among other things the Doxygen commented files used in order to generate the documentation. For further documentation files, the file called **documentation.h** is made to include more Doxygen documentation relative to the current example; it is also the entry point source of the Doxygen documentation. The file **complex_fft_example.c** is the implementation of the example. And finally, the file readme.html is the entry point of the whole automatically-generated documentation.

The directory **UTILS** contains all the utilities useful for the DSP library which includes the tools and the scripts. Here is its directory architecture:

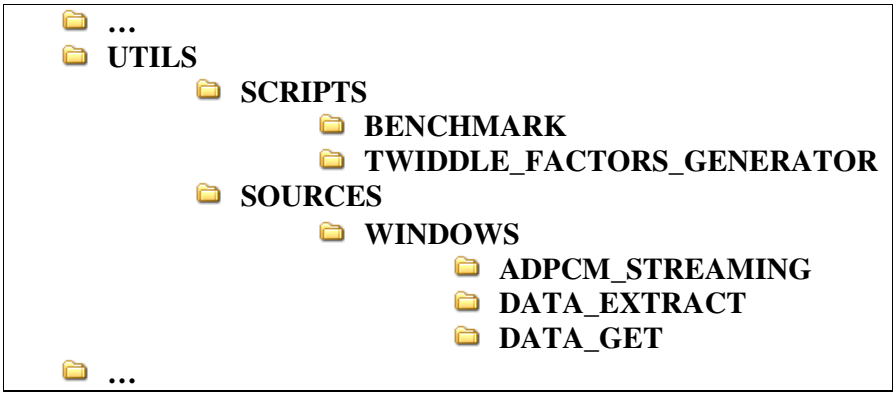


Figure 22: DSP library UTILS directory architecture

It is made of two subdirectories, a **SCRIPTS** directory which contains useful scripts sorted in directories and a **SOURCES** directory. This last directory currently contains only one directory defining the target one which this source is made for. Here only tools for Windows are available, therefore, only a **WINDOWS** directory is present in the **SOURCES** directory. In this directory is listed all the sources of the tools sorted by subdirectories. In the source directory, the tools' executables is also present.

The entire DSP library is set as a module of the SERVICES section of the Software Framework. This library is update on the framework with a CVS like tool called SVN.

2. *Complex FFT development*

With this example, I will describe the way I developed a new function.

2.1. Conception

At Fist, It took significant time for the conception part. This part includes the research of the documentation, the choice of the algorithm and the way I implemented it.

2.1.1. Documentation

At the beginning of this process, I tried to group all the documentation I could find on Internet. Then I extracted the important information and principally the different algorithms and their resource consumption. I have listed mainly 5 algorithms:

- The basic algorithm from the formula.
- Cooley-Tukey radix-N Decimation In Time (DIT)
- Cooley-Tukey split-radix Decimation In Time (DIT)
- Sande-Tukey radix-N Decimation In Frequency (DIF)
- Sande-Tukey split-radix Decimation In Frequency (DIF)

2.1.2. Algorithms

Before going further, let's introduce the complex Fast Fourier transforms:
The Fourier transform is a time domain to frequency domain mathematical transform. For any given continuous signal the Fourier transform provides a decomposition of the signal into the amplitudes and frequencies of a set of sine waves which would reproduce the original signal when summed. This property makes it easy to identify the frequencies, at which most of the signal strength is being transmitted.
For an N sample transform the Discrete Fourirer Transform (DFT) is defined by the formula:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \quad k = 0,1,\ldots, N-1$$

$$\text{where} \quad W_N = e^{-j(2\pi/N)}$$

Where x(n) is the sample at time index of n and X(k) is a vector of N values at frequency index k corresponding to the magnitude of the sine waves at resulting from the decomposition of the time indexed signal.
Examining the equation we can see that the calculation of the DFT would be of complexity $O(N^2)$. The direct implementation of this equation would require considerably hardware complexity and would be slow. So the basic algorithm, due to its poor performances was, out of question.

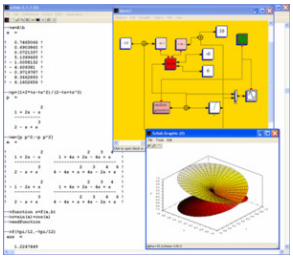
The N-radix algorithms are algorithms that compute N elements a time. Therefore, the highest N is, the fastest the algorithm runs but the less freedom you get. If you compute N elements a time that means you can compute an FFT on N^k elements only, which is very constraining when N is high.

The split-radix algorithms have the advantage of being a combination of both 2-radix and 4-radix algorithms in order to gain flexibility compared to the 4-radix algorithms and to gain speed in comparison with the 2-radix ones.

The difference between the "Decimate In Time" and the "Decimate In Frequency" algorithms is the direction in which you perform the algorithm. Indeed, to perform a radix algorithm, you have to change the elements' order of the input buffer, this at the beginning (DIT) or at the end (DIF). As I specified it, this function takes one buffer for the input and one for the output. Therefore, I could not modify the input buffer inside the function; I had to paste it first onto the output buffer before performing any computations. That's why I used the Decimate In Time (DIT) or Cooley-Tukey implementation.

For all of these reasons, I decided to implement the Cooley-Tukey split radix algorithm, but first I needed to implement the 2-radix and 4-radix DIT algorithms in order to mix them into the split-radix algorithm.

2.1.3. Algorithm conception



All of the following algorithms have been tested thanks to the freeware Scilab. Scilab is a numerical computational package developed by INRIA and "École nationale des ponts et chaussées" (ENPC) in France. It is a high level programming language, most of its functionality based around it being easy to condense many single computations into one line of code. It does this primarily by abstracting primitive data types to be functionally equivalent to matrices. It is similar in functionality to MATLAB.

2.1.3.1. Radix-2 DIT algorithm

The radix-2 algorithm is the simplest FFT algorithm. The decimation-in-time (DIT) radix-2 FFT recursively partitions a DFT into two half-length DFTs of the even-indexed and odd-indexed time samples. The outputs of these shorter FFTs are reused to compute many outputs, thus greatly reducing the total computational cost.
Before starting this recursion, the input buffer has to be initialized by changing the order of its elements. The indexes of its elements have to be bit-reversed. For example to perform a 16-element FFT, at the initialization, the second element of the input buffer (indexed 1 -> 0001 in binary format) will become the 9th element of the buffer (indexed 8 -> 1000 in binary format).
Then the recursion can start as describe by the following figure:

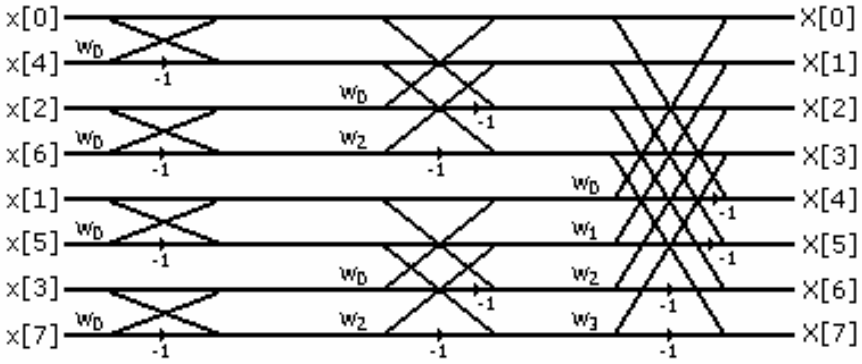


Figure 23: radix-2 DIT FFT algorithm

Each element of the recursion is called a butterfly and is calculated as follow:

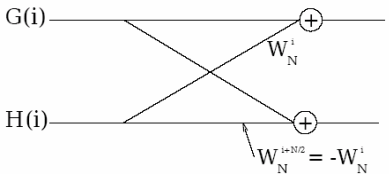


Figure 24: radix-2 DIT butterfly

In order to avoid future overflows in the summations from the butterflies, I scaled the data by 2 each time an addition is performed.

The complexity of this algorithm is **O(N Log2 N)**.

2.1.3.2. Radix-4 DIT algorithm

I will not describe this algorithm which is partly the same as the previous one for the exception that the butterflies are performed with four elements in input and not two as previously stated.
The complexity of this algorithm is about **O(3/4*N Log2 N)**.

2.1.3.3. Split-radix DIT algorithm

This algorithm is just a mix between the radix-2 and the radix-4 algorithms. This association will result into an "L" shaped butterfly:

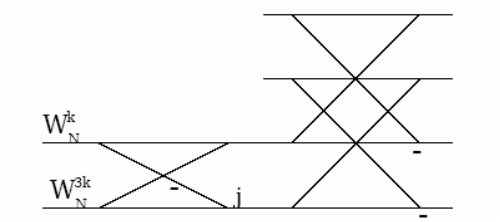


Figure 25: "L" shaped butterfly for the split-radix DIT algorithm

Then the butterflies will be performed as follow:

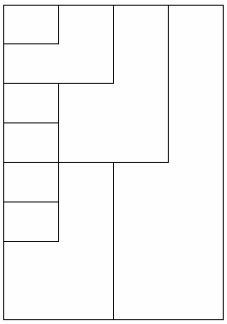


Figure 26: Shape of the split-radix DIT FFT

The complexity of this algorithm is about **O(2/3*N Log2 N)**.

The conception of this algorithm was difficult because I could not find any example on the internet about it and the explications of this algorithm were very vague.

2.1.4. Algorithm optimizations

Now it is time for the algorithmic optimization.
After several observations and tests, I realized that the split-algorithm is better because it uses fewer multiplications but compared to the optimization I performed on the radix-4 algorithm, it was not worth it to implement it in C language. Indeed, it is very complex to calculate the size of the next "L" shaped butterfly to be performed and even if the global complexity of this algorithm is smaller than the radix-4 algorithm, the complexity to switch between recursions and stages is much more important.
To confirm my observations, I developed both of the algorithms in C language and it appears that under 1024-point FFT, the radix-4 algorithm is much more efficient.
So I dropped the split-radix algorithm in order to optimize the 4-radix algorithm.

The first main observation I made about the radix-4 algorithm was that I could perform the first butterfly computing stage with the bit-reserve process at the initialization. In addition, many optimizations can be done at the first stage because the twiddle factors coefficients (the exp(-j*2*pi*w*k) in the basic formula) are all equal to 1. Therefore, it practically avoids the first stage processing when performing at the initialization which reduces the amount of cycles.
Another consequent algorithm optimization is made while performing the first recurrences of the current stage. In fact, during this computation it appears that all the twiddle factor coefficients are also equals to 1. The optimization is actually as efficient as the last one.
Here is the final algorithm used to implement the complex FFT function:

```
size = 1 << nlog

FOR r FROM 0 TO size-1 STEP 4 DO
        Butterfly_zero_only_real_and_bit_reversing(vect1, vect2, r)
END

FOR stage FROM 1 TO nlog/2 DO
        m = 4 ^ stage

        FOR r FROM 0 TO size-1 STEP m DO
                Butterfly_zero(vect1, r)
        END

        FOR j FROM 1 TO m / 4 - 1 DO
                Comput_twiddle_factors(e, e2, e3, j / m)

                FOR r FROM 0 TO size-1 STEP m DO
                        Butterfly(vect1, r, j, e, e2, e3)
                END
        END
END
```

2.2. Development

To develop this algorithm on the AVR32UC3 target, I first used the GCC compiler. I have already done some inline assembly code inside a C code with GCC using the macro "__asm__", so that was not a challenge.

2.2.1. Generic version

I first implemented the generic version to compute 16-bit fixed point data. The code was easy to make and to debug thanks to the previous implementation made on Scilab. I could easily compare the results at every execution point of my program. To debug the code I used the DDD interface for GDB debugger under cygwin.
Once this program returned satisfying results, I checked its robustness by trying to call the FFT function with borderline arguments due to the low range of fixed point basic types.

2.2.2. AVR32UC3 optimized version

The optimized version has been coded from beginning to end in assembly language in order to have the best performance possible. To be easy to compile for the user, I decided to code the entire function inside a C file, with the inline assembly code macro.
At the beginning of my development, I noticed a compiler bug appearing when using naked functions. I sent a Bugzilla report. Here is the bug's description I sent:

GCC – naked attribute

Even when you declare a function with arguments including the naked attribute, GCC compiler will add several instructions to the top of the function. This problem appears when you compile your program with no optimization level (-O0).

Example:

```
__attribute__ ((naked)) int add (int a, int b)
{
        __asm__ __volatile__ (
                "nop\n\t"
                );
}
```

Here is the assembly code of this function generated by GCC:

```
800018a2 <add>:
800018a2:       ef 4c ff fc     st.w r7[-4],r12
800018a6:       ef 4b ff f8     st.w r7[-8],r11
800018aa:       d7 03           nop
```

Thankfully, this bug was not an obstacle for the development and I could pass through it using the -O3 compilation option.

To code the assembly function and to test it, I first disassembled the object file generated by compiling the generic version of the algorithm previously coded. After about a full day's work, I finally commented on all of the code and simplified it to its minimum. Now I could start the optimization using DSP instructions.

2.2.2.1. Optimization

The AVR32UC3 has interesting features, principally for data load and storage. It can load a 64-bit data in 2 cycles with multiple addressing modes. So I used this instruction in order to load four 16-bit data a time. Then, DSP instructions allow you to manage 16-bit data into a 32-bit register which was perfect for computing a butterfly where 16-bit coefficients was previously loaded in 32-bit registers.
I performed many optimizations, using all the registers of the AVR32UC3 to limit at maximum load and store instructions.

Finally, my experiences have taught me some little tricks like avoiding as many jumps as possible because it takes a lot of time to compute, compared to another instruction, because the preprocessor cannot pre-fetch the next instruction until the jump is executed.
It is better to use MAC instructions using the same register as accumulator. Indeed, this is more efficient because once a MAC instruction is performed and is used with the same accumulator; it will execute the next MAC in only one cycle.

2.2.2.2. Tests

In order to test the robustness of the optimized version of this program, as for the generic function, I call the function with several borderline arguments. On small input vector length, the code is traced step by step to see exactly what the application does. This last step is done while programming the function and every result is compared to the Scliab version of the algorithm.

2.3. Example

With this function like for the others, I have provided an example showing a way to use the application programming interface. Each example is realized on the EVK1100 board and uses the USART interface to communicate with the user. This example performs a complex FFT and prints both the cycle count of the calculation and the result of the complex FFT. All examples are based on this example. Then I also made several tools, available in the UTILS directory, made to extract those data and to print them either on Scilab/Matlab or in another application I made.
I made three major tools to accomplish this task. The first one is a basic application made to retrieve data from a COM port. Another program is to extract data and format them into floating point format data. All those applications work with files for transferring data, then it is easily usable with any script languages like Bourne-shell or others variants. Finally, the last one is a little scope permitting to display a buffer issued from a file. I also made a script that makes it possible to use this scope as a real time scope, printing data received from the serial port.

Here is a screenshot of the scope receiving complex FFT result from the example:

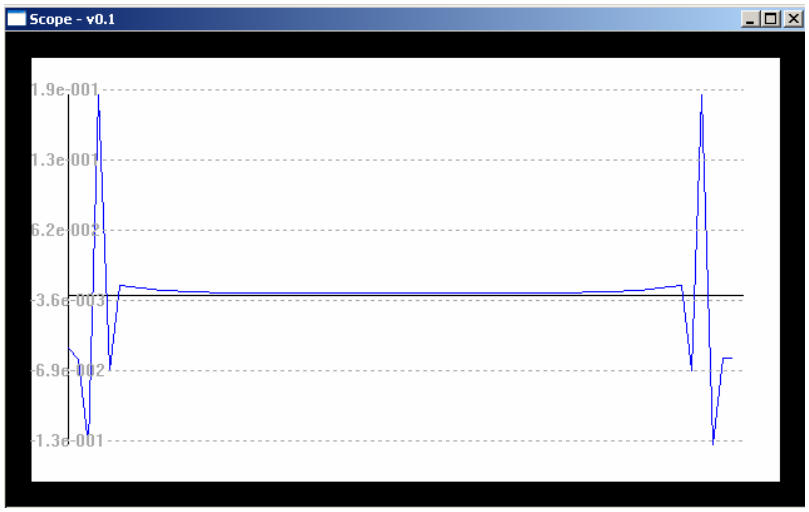


Figure 27: Screenshot of a 64-point complex FFT

2.4. Documentation

Each function is Doxygen documented.
I also made the profile of this function. The profile is a complete description of the function. Each profile of the DSP library is based on this one. Here is the profile I wrote for this specific function:

2.4.1. Description

This function computes a complex FFT from an input signal. It uses the Radix-4 "Decimate In Time" algorithm and does not perform a calculation "in place" which means that the input buffer has to be different from the output buffer.

2.4.1.1. Function prototype

void **dspXX_trans_complexfft**(
dspXX_complex_t **vect1*,
dspXX_t **vect2*,
int *nlog*);
where XX corresponds to the number of bits of a basic data element (i.e. 16 or 32).

2.4.1.2. Arguments

This function takes three parameters: the output buffer, the input buffer and a value corresponding to the size of those buffers.

- The output buffer (vect1) is a pointer on a complex vector of **2^nlog** elements.
- The input buffer (vect2) is a pointer on a real vector of **2^nlog** elements.
- The size argument (nlog) is in fact the base-2-logarithm of the size of the input vector. (nlog∈ [2, 4, 6, …, 28])

2.4.1.3. Algorithm

Following is the algorithm used to implement the radix-4 DIT complex FFT. The optimized version is based on this algorithm but can differ in certain points due to the instruction set of the target:

```
size = 1 << nlog

FOR r FROM 0 TO size-1 STEP 4 DO
        Butterfly_zero_only_real_and_bit_reversing(vect1, vect2, r)
END

FOR stage FROM 1 TO nlog/2 DO
        m = 4 ^ stage

        FOR r FROM 0 TO size-1 STEP m DO
                Butterfly_zero(vect1, r)
        END

        FOR j FROM 1 TO m / 4 - 1 DO
                Comput_twiddle_factors(e, e2, e3, j / m)
```

```
            FOR r FROM 0 TO size-1 STEP m DO
                Butterfly(vect1, r, j, e, e2, e3)
            END
        END
END
```

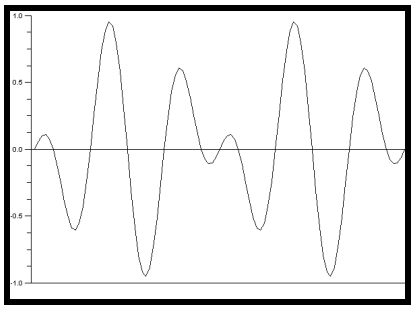
2.4.1.4. Notes

- **Interruptibility**: the code is interruptible.
- In-place computation is not allowed.
- This function uses a static twiddle factors table raw-coded in the file "*BASIC/TRANSFORMS/dspXX_twiddle_factors.h*". To generate these factors, you can use the script called "tf_gen.sci" and execute it with Scilab.
- To avoid overflowing values, the resulting vector amplitude is scaled by 2^nlog.
- All the vectors have to be 32-bit aligned.

2.4.2. Profiling

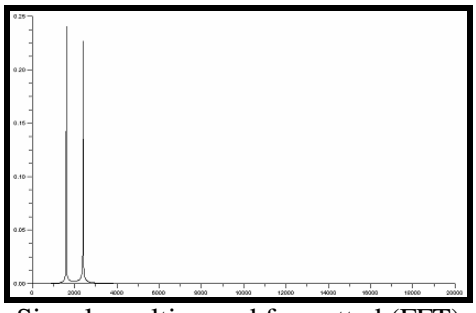
2.4.2.1. Benchmark routine

All these functions have been benchmarked on an avr32-uc3a0512 target. The programs have been compiled with avr32-gcc (4.0.2-atmel.1.0.0) with the –O3 optimization option and have been stored in FLASH memory. The fixed-point formats used are the Q1.15 format for the 16-bit data and the Q1.31 format for the 32-bit data.
The benchmark process was performed with the same input signal for all these functions, whereas the reference's signal computed with a mathematic tool using floating point.
The input signal is a combination of one sine and one cosine, the sine oscillating at 400 Hz and the cosine at 2 KHz. These signals have been multiplied and sampled at 40 KHz.



Input signal

Complex FFT



Signal resulting and formatted (FFT)

2.4.2.2. Result

Here are tables of the main values of the benchmark results. All these values correspond to the best performances of the functions and are obtained with different compilation options. For more information, please refer to the complete benchmark result table in annexes.

16-bit radix-4 D.I.T. complex FFT: generic

Concerned file path: */BASIC/TRANSFORMS/dsp16_complex_fft_generic.c*

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error		Lowest Algorithm's size in memory
			Amplitude average	Max. amplitude	
64-points	6,296	108.2us	1.58e-5	6.53e-5	1.1 Kbytes
256-points	33,723	578.0us	1.69e-5	8.80e-5	1.3 Kbytes
1024-points	169,006	2.90ms	1.67e-5	12.31e-4	2.0 Kbytes
4096-points	812,321	13.90ms	1.52e-5	14.60e-4	5.0 Kbytes

More details on Table XXX in annexes

16-bit radix-4 D.I.T. complex FFT: avr32-uc3 optimized

Concerned file path: */BASIC/TRANSFORMS/dsp16_complex_fft_avr32uc3.c*

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error		Lowest Algorithm's size in memory
			Amplitude average	Max. amplitude	
64-points	2,611	44.4 us	1.63e-5	6.53e-5	710 bytes
256-points	13,661	232.2 us	1.68e-5	7.46e-5	902 bytes
1024-points	67,671	1.15 ms	1.69e-5	1.02e-4	1.6 Kbytes
4096-points	322,897	5.49 ms	1.58e-5	1.18e-4	4.6 Kbytes

Warning: this function is only compatible with Q1.15 numbers.

Note: this function needs 72 bytes of memory for the stack.

More details on Table XXX in annexes

32-bit radix-4 D.I.T. complex FFT: generic

Concerned file path: */BASIC/TRANSFORMS/dsp32_complex_fft_generic.c*

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error		Lowest Algorithm's size in memory
			Amplitude average	Max. amplitude	
64-points	13,206	225.2us	6.0e-10	5.7e-9	2.0 Kbytes
256-points	74,297	1.27us	3.0e-10	4.8e-9	2.4 Kbytes
1024-points	383,212	6.53ms	3.0e-10	6.1e-9	3.9 Kbytes

More details on Table XXX in annexes

2.5. New instructions

During the development of this function, I noticed that it would be interesting if the AVR32UC3 had two more instructions in its instruction set. The architect of this microcontroller came from Norway to Nantes for a meeting in March and asked the Application Laboratory if we could do some implementations for the new revision of this microcontroller to make it more efficient. I talked to him about the new instructions he could add and he accepted. Then after a while he contacted me with a software solution that could work instead of adding those instructions. I demonstrated to him that indeed it could work but only in specific cases, which of course was not applicable in the FFT functions. Finally, he decided not to add those instructions in the next revision because of all the constraints it would have generated (different software to ensure an optimized version for each revision, etc).

Here is a description of those two new instructions and a reason why it would be advantageous to add them into the architecture of the AVR32UC3 core:

- **adddiv2hh.w** Rd, Rx<part>, Ry<part>
  Operation:
  Rd <- (Rx<part> + Ry<part>) >> 1

- **subdiv2hh.w** Rd, Rx<part>, Ry<part>
  Operation:
  Rd <- (Rx<part> - Ry<part>) >> 1

These operations are really useful when you are manipulating numbers and when you don't want them to overflow. It is even more useful when those numbers are fixed point defined and are Q1.15 formatted numbers because their range is very limited (from -1 to 1). Therefore, those types of numbers can easily overflow when you add or subtract two of them.

Example:

When you compute a FFT with those kinds of formatted numbers, you have to divide the result by 2 each time you add or subtract numbers. For a 64-points 4-radix complex FFT, you have 3 stages. Each stage is composed of 16 "4-radix-buterflies". So you have to compute 3*16 = 48 butterflies. Each of those butterflies is composed of 16 Q1.15 additions/subtractions, which means with the classical instruction set, you have to perform 8 bit-shifting (because you re-use additions to perform other additions).
So the total number of additions/subtractions is 16*48=768. Therefore, with such instructions, you can economize approximately 8*48 = 384 cycles in your program (because a shift-right takes 1 cycle to be executed).

It takes 2952 cycles to perform an optimized 64-points complex FFT (with DSP_OPTI_ACC algorithm optimization) while it would take only 2632 cycles to perform the same FFT with the same accuracy with these new instructions (this evaluation has been performed by deleting the "asr #, 1" after an addition or a subtraction in a FFT routine).

Comparatively, STMicroelectronics has this feature in its instructions ADD and SUB:

ADD R0, R1, R2, ASR#1
SUB R0, R1, R2, ASR#1

2.6. Performances comparison

The following graphic shows a performances' comparison of multiple complex FFT computations for different sizes on several targets. The Y axis corresponds to the cycle count of the computation and, therefore, the less it is the more efficient the complex FFT will be. The X axis is the number of point the complex FFT has been benchmarked with.

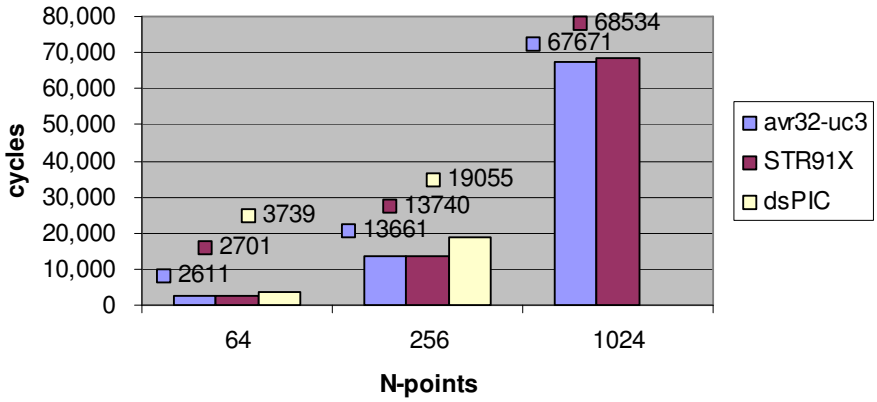


Figure 28: Complex FFT performances comparison

The STR91X and the dsPIC are equivalent microcontrollers as the AVR32UC3.

3. *Others basic functions development*

I will not describe the whole process I used to develop the following functions. This process is similar to the one I used for the complex FFT.
I developed other useful basic DSP functions like FIR and IIR Filters, convolution functions, sine and cosine operators for fixed point data. For more details on the functions, refer to the "Profile" document in annexes.
Here are some comparatives with the concurrency:

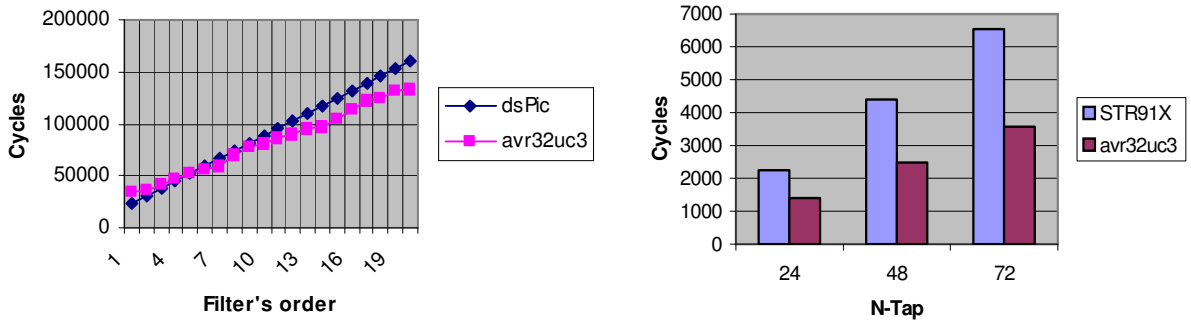


Figure 29: IIR performances comparison

4. *ADPCM streaming player*

I made a full example of a streaming IMA/DVI ADPCM decoder.
It uses the USART to receive data blocks (compressed in IMA/DVI ADPCM format) and two mixed PWM to play through a 16 bits precision sample sound.

4.1. Conception

The code is based on two interrupts, the PWM interrupt and the PDCA interrupt.

At first, the program will send a 'S' character on the USART line to tell the host that it is ready to start the data acquisistion. The host will first send initialization parameters like the size of a block and the bit rate of the sound. Once the application acknowledges this data, the data block acquisition can start.
At the begining of the process, the application fills all its buffers before playing any data. Then, it can start playing a data block on the PWMs. Each time a data block is read, it will check if the next one is available, if not it will wait until it is completly filled. Equally, each time a data block is filled, it will check if the next data block as already been played or not. If not it will wait or else it will start the next data block acquisition process.
You will find in section "Protocol" the whole protocol description of the data block transfer.

4.2. Hardware Interface

Here is the hardware interface used to mix the two PWM, to make them act as a 16-bit-DAC.

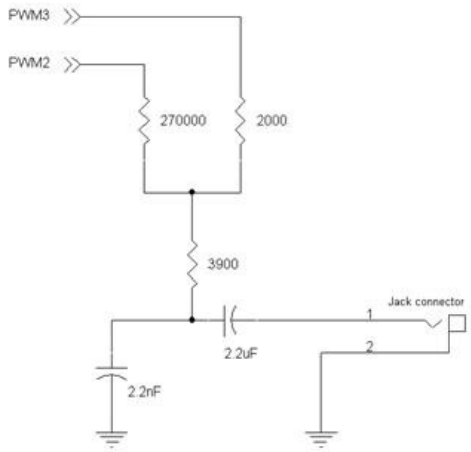


Figure 30: PWM to DAC mixer

4.3. Protocol

Here is a description of the protocol used to communicate between the PC and the microcontroller.
To initialize the transfer, the target has to send the character 'S' through the USART and then the host will send a block of 8 bytes described as follow:

Length of the data block (4 bytes)				Sample rate (4 bytes)			
3	2	1	0	3	2	1	0

Where 3 corresponds to the most significant byte of the data.

Figure 31: Initialization block

This block will fix the length of the following blocks used to transfer the IMA/DVI ADPCM data (this length is called "block_size") and will also specify the sample rate of the sound.
Then the host waits until a byte value of 1 is sent to the line. Each time it recieves this character, it will send to the target a block of (block_size + 4) bytes of IMA/DVI ADPCM data containing the sound samples. The first 4 bytes of this block contain the "Predicted Value" and the "Step Index" values used in the IMA/DVI ADPCM algorithm to compute the following data. The following data are the sound samples, coded in IMA/DVI ADPCM on 4 bits per sample.

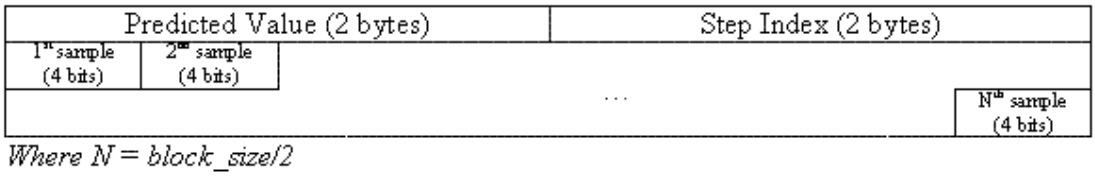


Figure 32: Block format

I also developed a tool in C language that permits the user to send IMA/DVI ADPCM data through the serial port of a computer. It takes in parameter the IMA/DVI ADPCM encoded wav file you want to transfer and if the file is valid, it will cut the file into blocks to send them through the serial port. The serial port used for transferring the data is the COM1 and is configured as follow: 57600 bauds, 8 bits, no parity and 1 stop bit.

```
IMA/DVI ADPCM decoder
File name: D:\sample_ima_adpcm_8kHz.wav
Number of channels: 1
Sample rate: 8000 Hz
Total average data rate: 4055
Block alignment: 256 bytes
Number of bits per sample: 4 bits
Number of samples per channel per Block: 505 samples
opening serial port COM1...      [ OK ]
Waiting for serial communication with the interface...  [ OK ]
Sending initialization parameters.
[=========                       ] 76104 bytes
```

Figure 33: IMA/DVI ADPCM streaming tool

4.4. CPU charge

Here is the CPU charge of the decoding process for a sound sampled at 8000 Hz and with a target running at 48 MHz:

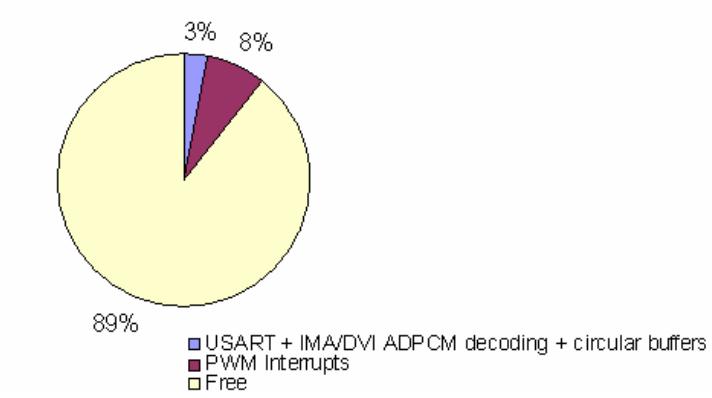


Figure 34: CPU charge of the streaming IMA/DVI ADPCM player using PDCA

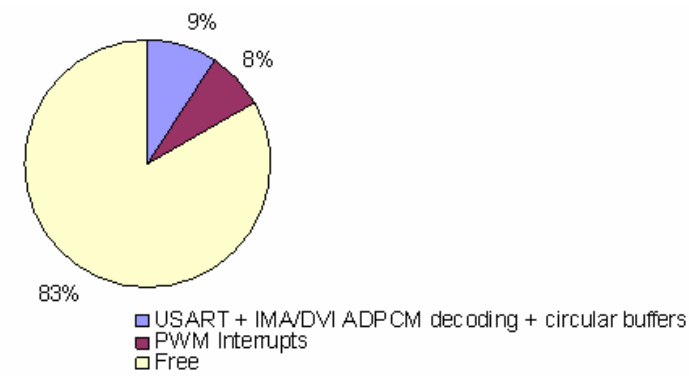


Figure 35: CPU charge of the streaming IMA/DVI ADPCM player using USART interrupts

5. Benchmarks

5.1. Script automation

For example, in order to perform a full benchmark for the complex FFT, I had to report a benchmark on the "Profile" document in annexes and I had to recompile 64 times the program. Indeed, to make a full benchmark, I needed 4 different size input signals using the 4 different algorithm optimization options with data placed on FLASH or on SRAM with different speeds (because at a certain speed, FLASH access required a wait state). Once a program was compiled with the right options, I had to extract the cycle count of the function, the error of the result (compared to a Scilab equivalent function) and the size of the function. Therefore, a full benchmark would take me about one day of work to get all of the results and that's why I decided to create a benchmark automation script in order to be more efficient.
I developed the script in Bourne shell language and it uses two others applications I made, the DataGet and DataExtract programs. The first one is to retrieve data from the COM port and the second is to extract them.
The script is made to be the most configurable as possible. You can specify 4 different defines with an unlimited list of values that the program will include when compiling.

Here is a diagram showing the process used by the script to perform the full benchmark:

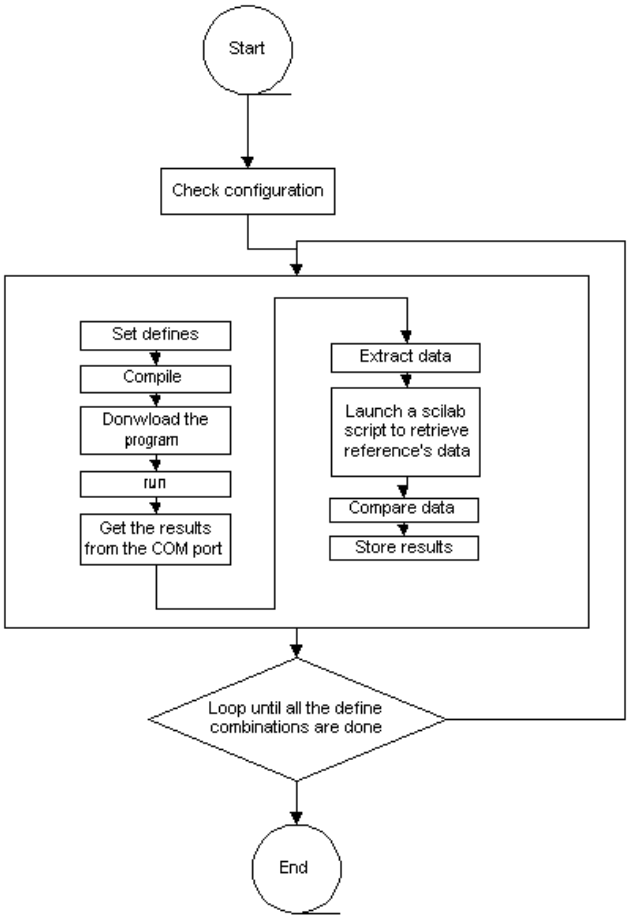


Figure 36: Automation benchmark process diagram

The documentation of this script is available in annexes of this document.

5.2. TI Bench

The marketer asked me to perform a benchmark provided by TI. I accomplished it and you can find the results in annexes, in the document titled "TI Bench results".
Here is a small report I sent on Bugzilla to explain what I noticed thanks to my experiments.

After performing some benchmarks provided by TI in C language, it appears that the avr32uc3 is not as optimized as it should be for GCC compiler. Indeed, we do better or about equal results than the ARM7 on every function tests except the one that include floating-point computations. Here is a paper concerning this benchmark: http://cores63.nto.atmel.com/svnavr32/apps/benchmarks/SignalProcessing/TIbenchs/MSP430_Competitive_Benchmarking.pdf And here are the results of the avr32uc3 compared to several microcontrollers and especially the ARM7 core: http://cores63.nto.atmel.com/svnavr32/apps/software/swlib/trunk/SERVICES/DSPLIB/BENCHMARKS/TI%20BENCH%20-%20results.pdf

Summary of revelant results: (Code compiled with the 4.0.2-100 version of avr32-gcc with full optimization)

Benchmark:

Dhrystone:	UC3: 25,283 cycles	ARM7: 52,352 cycles
→ FIR Filter:	UC3: **109,559** cycles	ARM7: 33,114 cycles
→ Floating-point Math:	UC3: **717** cycles	ARM7: 187 cycles
Whetstone:	UC3: **59,424** cycles	ARM7: 60,444 cycles

After assembly code analysis, it shows clearly that GCC for UC3 is computing floating-point data, the results are poor. The problem seems to come the way the following intrinsic functions for gcc uc3 are assembly coded:
__avr32_s32_to_f32
__addsf3
__avr32_f32_add
__avr32_f64_cmp_lt
__avr32_f64_mul
__avr32_f64_div
__avr32_f64_cmp_ge
__avr32_f32_sub
__avr32_f64_to_s32
__avr32_f64_to_u32
__avr32_s32_to_f64
__avr32_f64_sub
__avr32_f64_add
__avr32_u32_to_f64
__avr32_f64_cmp_eq
__avr32_f32_mul
__avr32_f32_div
The Whetstone is not as efficient as the Dhrystone because it also computes floating-point operations.

Remark: the ARM7 has no FPU like the avr32uc3.

Conclusion

This document was a brief presentation about the training subject. The objective was to develop an optimized digital signal processing library for the AVR32UC3 fully document my work and to give its time constraints.

This training placement was an occasion for Atmel and for me to improve our knowledge and to test some methodologies. I benefited from this experience in two areas:

- **Project management:**
  I increased my skills in project management. It was the first time I worked alone on such an important project and it was not easy to estimate the time of each task. Thanks to these conditions, I realized that it is necessary to describe the entire project's tasks well and that a planning is always changing, due to unplanned urgent birth of activities.
- **Technical:**
  → This training allowed me to see an engineering job from another point of view. Now I understand how product specifications can be realized with the up most precision and how interactivity must be created between all the project members (engineers, marketers, etc).
  → This training allowed me to develop my skills in benchmarking and CPU charge distributions.
  → I developed competences in algorithm optimization, code optimization, assembly language and in compilation process comprehension. This training placement also allowed me to improve my C programming knowledge.
  → Furthermore I studied standard documentation and became aware of the difficulty in reading these types of documents.
  → My internship gave me the occasion to work on an early release of a recent microcontroller.
  → Finally, I improved my signal processing knowledge.

For Atmel, I created from nothing an optimized DSP library compatible with their new product series, the AVR32UC3. I also developed a new driver and added in the driver section of the software framework. Finally, I made several benchmarks on the MCU to measure its performances and also in order to validate and to be competitive with the competition.

REFERENCES

AT32UC3A ADVANCED DATASHEET
Atmel intern and confidential document
AVR32 ARCHITECTURE DOCUMENT
Atmel intern and confidential document
AVR32UC TECHNICAL REFERENCE MANUAL
Atmel intern and confidential document
CIRCUIT CELLAR, SIGNAL PROCESSING
Number 194, September 2006
TMS320C64X+ DSP LITTLE-ENDIAN DSP LIBRARY PROGRAMMER'S REFERENCE
http://focus.ti.com/lit/ug/sprueb8/sprueb8.pdf
STR91X DSP LIBRARY (DSPLIB)
http://www.st.com/stonline/products/literature/um/12815.pdf
SIGLIB, SIGNAL PROCESSING LIBRARY, FUNCTION REFERENCE MANUAL
http://www.numerix-dsp.com/docs/ref.pdf
A GUIDE TO THE DSP LIBRARY
http://www.eagleware.com/pdf/Dsp.pdf
DSPIC DSC, DSP LIBRARY
http://ww1.microchip.com/downloads/en/DeviceDoc/01033B 28.pdf
FFT BACKGROUND
http://www.ece.uvic.ca/499/2004a/group05/html/background.html
TRANSFORMEE DE FOURIER RAPIDE
http://magphy.ujf-grenoble.fr/orbis/files/cours/2002/maitrise/cours_signal/Cours4.pdf
FASTEST FOURIER TRANSFORM IN THE WEST
http://www.fftw.org/index.html
FFT ALGORITHMS
http://www.cbrc.jp/~tominaga/translations/gsl/fftalgorithms.pdf
FIXED POINT REPRESENTATION AND FRACTIONAL MATH
http://www.superkits.net/whitepapers/Fixed Point Representation & Fractional Math.pdf
FIXED-POINT ARITHMETIC: AN INTRODUCTION
http://www.digitalsignallabs.com/fp.pdf
FAST CONVOLUTION
http://inst.eecs.berkeley.edu/~ee123/sp07/resources/m12022.pdf
ADAPTIVE ECHO CANCELLATION, THE NLMS ALGORITHM
Not available anymore.
FILTRE DE WIENER
http://www.tsi.enst.fr/~charbit/cours/TSMAp/tsmap-CHwienerlms.pdf
AVR336: ADPCM DECODER
http://www.atmel.com/dyn/resources/prod_documents/doc2572.pdf

ANNEXES

DSP library functions' profile

TABLE OF CONTENTS

Radix-4 decimate in time complex FFT

Description

This function computes a complex FFT from an input signal. It uses the Radix-4 "Decimate In Time" algorithm and does not perform a calculation "in place" which means that the input buffer has to be different from the output buffer.

Function prototype

void **dspXX_trans_complexfft**(
dspXX_complex_t **vect1*,
dspXX_t **vect2*,
int *nlog*);

where XX corresponds to the number of bits of a basic data element (i.e. 16 or 32).

Arguments

This function takes three parameters: the output buffer, the input buffer and a value corresponding to the size of those buffers.

- The output buffer (vect1) is a pointer on a complex vector of **2^nlog** elements.
- The input buffer (vect2) is a pointer on a real vector of **2^nlog** elements.
- The size argument (nlog) is in fact the base-2-logarithm of the size of the input vector. (nlog∈ [2, 4, 6, …, 28])

Algorithm

Following is the algorithm used to implement the radix-4 DIT complex FFT. The optimized version is based on this algorithm but can differ in certain points due to the instruction set of the target:

```
size = 1 << nlog

FOR r FROM 0 TO size-1 STEP 4 DO
        Butterfly_zero_only_real_and_bit_reversing(vect1, vect2, r)
END

FOR stage FROM 1 TO nlog/2 DO
        m = 4 ^ stage

        FOR r FROM 0 TO size-1 STEP m DO
                Butterfly_zero(vect1, r)
        END

        FOR j FROM 1 TO m / 4 - 1 DO
                Comput_twiddle_factors(e, e2, e3, j / m)

                FOR r FROM 0 TO size-1 STEP m DO
                        Butterfly(vect1, r, j, e, e2, e3)
                END
        END
```

END

Notes

- **Interruptibility**: the code is interruptible.
- In-place computation is not allowed.
- This function uses a static twiddle factors table raw-coded in the file "*BASIC/TRANSFORMS/dspXX_twiddle_factors.h*". To generate those factors, you can use the script called "tf_gen.sci" and execute it with Scilab.
- To avoid overflowing values, the resulting vector amplitude is scaled by 2^nlog.
- All the vectors have to be 32-bit aligned.

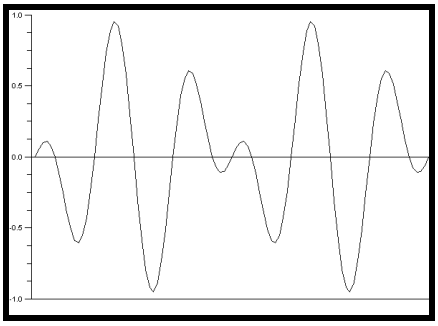
Benchmark

Benchmark routine

All these functions have been benchmarked on an avr32-uc3a0512 target. The programs have been compiled with avr32-gcc (4.0.2-atmel.1.0.0) with the –O3 optimization option and have been stored in FLASH memory. The fixed-point format used is the Q1.15 format for the 16-bit data and the Q1.31 format for the 32-bit data.

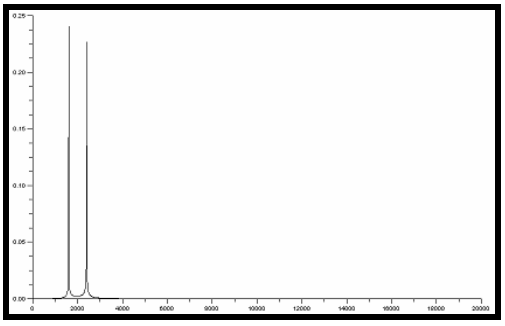
The benchmark process has been performed with the same input signal for all those functions and compared with a reference's signal computed with a mathematic tool using floating point.

The input signal is a combination of one sine and one cosine. The sine oscillating at 400Hz and the cosine at 2KHz. Those signals have been multiplied and sampled at 40KHz.



Input signal

Complex FFT



Signal resulting and formatted (FFT)

Result

Here are tables of the main values of the benchmark results. All those values correspond to the best performances of the functions and are obtained with different compilation options. For more information, please refer to the complete benchmark result table in annexes.

16-bit radix-4 D.I.T. complex FFT: generic

Concerned file path: */BASIC/TRANSFORMS/dsp16_complex_fft_generic.c*

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error		Lowest Algorithm's size in memory
			Amplitude average	Max. amplitude	
64-points	6,296	108.2us	1.58e-5	6.53e-5	1.1 Kbytes
256-points	33,723	578.0us	1.69e-5	8.80e-5	1.3 Kbytes
1024-points	169,006	2.90ms	1.67e-5	12.31e-4	2.0 Kbytes
4096-points	812,321	13.90ms	1.52e-5	14.60e-4	5.0 Kbytes

More details on Table 1.1.1 in annexes

16-bit radix-4 D.I.T. complex FFT: avr32-uc3 optimized

Concerned file path: */BASIC/TRANSFORMS/dsp16_complex_fft_avr32uc3.c*

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error		Lowest Algorithm's size in memory
			Amplitude average	Max. amplitude	
64-points	2,611	44.4 us	1.63e-5	6.53e-5	710 bytes
256-points	13,661	232.2 us	1.68e-5	7.46e-5	902 bytes
1024-points	67,671	1.15 ms	1.69e-5	1.02e-4	1.6 Kbytes
4096-points	322,897	5.49 ms	1.58e-5	1.18e-4	4.6 Kbytes

Warning: this function is only compatible with Q1.15 numbers.
Note: this function needs 72 bytes of memory for the stack.

More details on Table 1.1.2 in annexes

32-bit radix-4 D.I.T. complex FFT: generic

Concerned file path: */BASIC/TRANSFORMS/dsp32_complex_fft_generic.c*

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error		Lowest Algorithm's size in memory
			Amplitude average	Max. amplitude	
64-points	13,206	225.2us	6.0e-10	5.7e-9	2.0 Kbytes
256-points	74,297	1.27us	3.0e-10	4.8e-9	2.4 Kbytes
1024-points	383,212	6.53ms	3.0e-10	6.1e-9	3.9 Kbytes

More details on Table 1.2.1 in annexes

Convolution

Description

This function performs a linear convolution between two discrete sequences.

Function prototype

void **dspXX_vect_conv**(
dspXX_t *_vect1_,
dspXX_t *_vect2_,
int vect2_size_,
dspXX_t *_vect3_,
int _vect3_size_);

where XX corresponds to the number of bits of a basic data element (i.e. 16 or 32).

Arguments

This function takes five parameters: the output buffer, the two discrete sequences and their respective sizes.

- The output buffer (vect1) is a pointer on a real vector of **(vect2_size + vect3_size - 1)** elements.
- The first input buffer (vect2) is a pointer on a real vector of **vect2_size** elements.
- The first size argument (vect2_size) is the length of the first input buffer (vect2_size∈ [8, 9, 10, …]).
- The second input buffer (vect3) is a pointer on a real vector of **vect3_size** elements.
- The second size argument (vect3_size) is the length of the second input buffer (vect3_size ∈ [8, 9, 10, …]).

Requirements

This function requires 3 modules:

Module name	Function name	Concerned file path
Zero Padding	dspXX_vect_zeropad	*/BASIC/VECTORS/zero_padding.c*
Copy	dspXX_vect_copy	*/BASIC/VECTORS/copy.c*
Partial Convolution	dspXX_vect_convpart	*/BASIC/VECTORS/convolution_partial.c*

The output buffer of the function has to have at least a length of **N + 2*M – 2** elements because of intern computations, where N is the length of the largest input buffer and M, the length of the smallest input buffer.

Algorithm

Following is the algorithm used to implement the convolution product. The optimized version is based on this algorithm but can differ in certain points due to the instruction set of the target:

IF vect2_size **>=** vect3_size **THEN**

vect1 = | **0 0 0 0 … 0 0 0 0** | **vect2** | **0 0 0 0 … 0 0 0 0** |

vect3_size – 1 *vect2_size* *vect3_size – 1*

Partial_convolution(vect1, vect1, vect2_size + 2*(vect3_size – 1), vect3, vect3_size)

ELSE

vect1 = | **0 0 0 0 … 0 0 0 0** | **vect3** | **0 0 0 0 … 0 0 0 0** |

vect2_size – 1 *vect3_size* *vect2_size – 1*

Partial_convolution(vect1, vect1, vect3_size + 2*(vect2_size – 1), vect2, vect2_size)

END

Notes

- **Interruptibility**: the code is interruptible.
- Due to its implementation, the dsp16-avr32-uc3 optimized version of the FIR requires a length of 4*m elements for the largest input discrete sequence and the output buffer (vect1) has to have a length of 4*n elements to avoid overflows.
- The input discrete sequences have to be scaled to avoid overflowing values.
- All the vectors have to be 32-bit aligned.

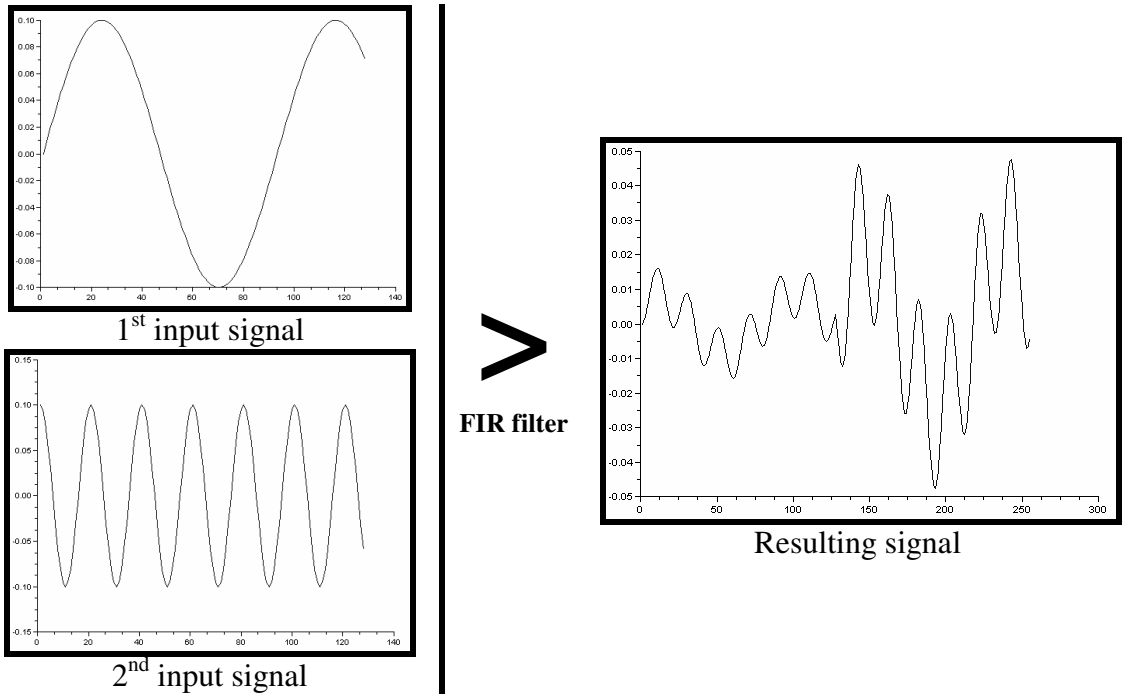
Benchmark

Benchmark routine

All these functions have been benchmarked on an avr32-uc3a0512 target. The programs have been compiled with avr32-gcc (4.1.2-atmel.1.0.0) with the –O3 optimization option and have been stored in FLASH memory. The fixed-point format used is the Q1.15 format for the 16-bit data and the Q1.31 format for the 32-bit data.

The benchmark process has been performed with the same input signal and impulse response for all those functions and compared with a reference's signal computed with a mathematic tool using floating point.

The first input signal is a sine oscillating at 433Hz and the second input signal is a cosine oscillating at 2KHz. Those signals are sampled at 40KHz.



Result

Here are tables of the main values of the benchmark results. All those values correspond to the best performances of the functions and are obtained with different compilation options. For more information, please refer to the complete benchmark result table in annexes.

Concerned file path: */BASIC/VECTORS/convolution.c*

16-bit Convolution: generic

Algorithm's size in memory: 2.2 Kbytes.
Length of the first input signal: 64 elements.

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
32-points	23,524	408.5us	2.0e-5	4.5e-5
64-points	57,757	1.00ms	1.8e-5	4.7e-5
128-points	86,752	1.50ms	1.8e-5	4.4e-5
256-points	144,736	2.51ms	1.5e-5	4.8e-5

More details on Table 2.1.1 in annexes

16-bit Convolution: avr32-uc3 optimized

Algorithm's size in memory: 950 bytes.
Length of the first input signal: 64 elements.

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
32-points	8,248	151.2us	2.0e-5	4.5e-5
64-points	19,087	349.1us	1.8e-5	4.7e-5
128-points	28,532	521.8us	1.8e-5	4.4e-5
256-points	47,412	866.9us	1.5e-5	4.8e-5

More details on Table 2.1.2 in annexes

32-bit Convolution: generic

Algorithm's size in memory: 3.3 Kbytes.
Length of the first input signal: 64 elements.

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
32-points	42,572	729.8us	0.4e-9	2.1e-9
64-points	109,179	1.87ms	0.4e-9	1.7e-9
128-points	163,968	2.81ms	0.5e-9	1.6e-9
256-points	273,536	4.69ms	0.6e-9	2.7e-9

More details on Table 2.2.1 in annexes

32-bit Convolution: avr32-uc3 optimized

Algorithm's size in memory: 1.5 Kbytes.
Length of the first input signal: 64 elements.

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
32-points	19,958	340.2us	0.5e-9	2.1e-9
64-points	50,501	860.4us	0.5e-9	1.7e-9
128-points	75,722	1.29ms	0.6e-9	2.4e-9
256-points	126,154	2.15ms	0.7e-9	2.7e-9

More details on Table 2.2.2 in annexes

FIR Filter *(alias Partial Convolution)*

Description

This function computes a real FIR filter using the impulse response of the desire filter onto a fixed-length signal.

Function prototype

void **dspXX_filt_fir**(
dspXX_t **vect1*,
dspXX_t **vect2*,
int *size*,
dspXX_t **h*,
int *h_size*);

void **dspXX_vect_convpart**(
dspXX_t **vect1*,
dspXX_t **vect2*,
int vect2*_size*,
dspXX_t **vect3*,
int *vect3_size*);

where XX corresponds to the number of bits of a basic data element (i.e. 16 or 32).

Arguments

This function takes five parameters: the output buffer, the input buffer, its size, the impulse response of the filter and its size.

- The output buffer (vect1) is a pointer on a real vector of **(size - h_size + 1)** elements.
- The input buffer (vect2) is a pointer on a real vector of **size** elements.
- The size argument (size) is the length of the input buffer (size∈ [4, 8, 12, …]).
- The impulse response of the filter (h) is a pointer on a real vector of **h_size** elements.
- The size argument (h_size) is the length of the impulse response of the filter (h_size∈ [8, 9, 10, …]).

Requirements

This function requires one module:

Module name	Function name	Concerned file path
Partial Convolution	dspXX_vect_convpart	*/BASIC/VECTORS/convolution_partial.c*

Algorithm

Following is the algorithm used to implement the FIR filter. The optimized version is based on this algorithm but can differ in certain points due to the instruction set of the target:

```
FOR j FROM 0 TO size - h_size + 1 DO
        sum = 0

        FOR i FROM 0 TO h_size DO
                sum += vect2[i] * h[h_size - i - 1]
        END

        vect1[j] = sum >> DSPXX_QB
```

END

Notes

- **Interruptibility**: the code is interruptible.
- Due to its implementation, for the dsp16-avr32-uc3 optimized version of the FIR, the output buffer (vect1) has to have a length of 4*n elements to avoid overflows.
- The impulse response of the filter has to be scaled to avoid overflowing values.
- All the vectors have to be 32-bit aligned.

Benchmark

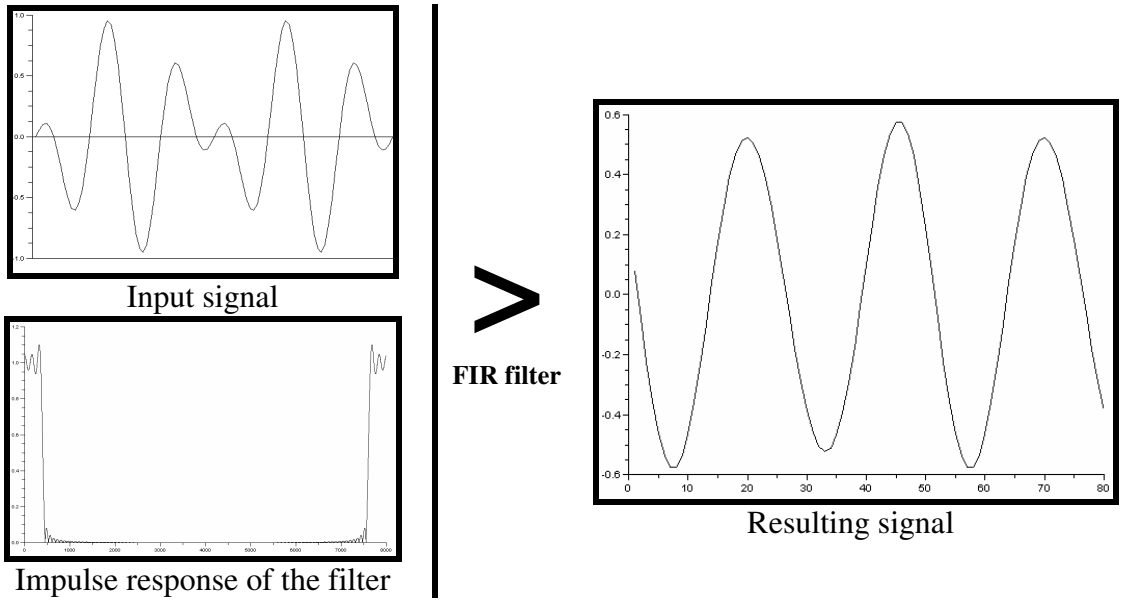
Benchmark routine

All these functions have been benchmarked on an avr32-uc3a0512 target. The programs have been compiled with avr32-gcc (4.1.2-atmel.1.0.0) with the –O3 optimization option and have been stored in FLASH memory. The fixed-point format used is the Q1.15 format for the 16-bit data and the Q1.31 format for the 32-bit data.

The benchmark process has been performed with the same input signal and impulse response for all those functions and compared with a reference's signal computed with a mathematic tool using floating point.

The input signal is a combination of one sine and one cosine. The sine oscillating at 400Hz and the cosine at 2KHz. Those signals have been multiplied and sampled at 40KHz.

The impulse response describes a low-pass filter with a cutoff frequency equal to 400Hz.



Result

Here are tables of the main values of the benchmark results. All those values correspond to the best performances of the functions and are obtained with different

compilation options. For more information, please refer to the complete benchmark result table in annexes.

16-bit FIR filter: generic

Concerned file path: */BASIC/VECTORS/dsp16_convpart_generic.c*
Algorithm's size in memory: 2.0 Kbytes.
Number of Taps: 24.

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
64-points	7,424	128.0us	2.27e-5	9.46e-5
256-points	41,793	720.0us	2.22e-5	9.46e-5
512-points	87,617	1.51ms	2.23e-5	9.46e-5
1024-points	179,265	3.09ms	2.21e-5	9.46e-5

More details on Table 3.1.1 in annexes

16-bit FIR filter: avr32-uc3 optimized

Concerned file path: */BASIC/VECTORS/dsp16_convpart_avr32uc3.c*
Algorithm's size in memory: 770 bytes.
Number of Taps: 24.

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
64-points	2,439	44.3us	2.27e-5	9.46e-5
256-points	12,712	230.7us	2.22e-5	9.46e-5
512-points	26,408	479.2us	2.23e-5	9.46e-5
1024-points	53,800	976.3us	2.21e-5	9.46e-5

More details on Table 3.1.2 in annexes

32-bit FIR filter: generic

Concerned file path: */BASIC/VECTORS/dsp32_convpart_generic.c*
Algorithm's size in memory: 3.1 Kbytes.
Number of Taps: 24.

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
64-points	13,984	239.4us	2.1e-9	1.24e-8
256-points	79,073	1.35ms	2.3e-9	1.74e-8
512-points	165,857	2.84ms	2.6e-9	2.31e-8
1024-points	339,425	5.81ms	3.7e-9	2.84e-8

More details on Table 3.2.1 in annexes

32-bit FIR filter: avr32-uc3 optimized

Concerned file path: */BASIC/VECTORS/dsp32_convpart_avr32uc3.c*
Algorithm's size in memory: 1.3 Kbytes.
Number of Taps: 24.

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
64-points	6,479	110.2us	2.1e-9	1.24e-8
256-points	36,432	619.0us	2.3e-9	1.24e-8
512-points	76,368	1.30ms	2.6e-9	2.31e-8
1024-points	156,240	2.65ms	3.7e-9	2.84e-8

More details on Table 3.2.2 in annexes

IIR Filter

Description

This function computes a real IIR filter using the impulse response of the desire filter onto a fixed-length signal.

Function prototype

void **dspXX_filt_iir**(
dspXX_t *vect1,
dspXX_t *vect2,
int *size*,
dspXX_t *num,
int *num_size,*
dspXX_t *den,
int *den_size,*
int *prediv*);

where XX corresponds to the number of bits of a basic data element (i.e. 16 or 32).

Arguments

This function takes five parameters: the output buffer, the input buffer, its size, the coefficients of the filter, theirs sizes and a coefficient's predivisor.

- The output buffer (vect1) is a pointer on a real vector of **(size - num_size + 1)** elements.
- The input buffer (vect2) is a pointer on a real vector of **size** elements.
- The size argument (size) is the length of the input buffer (size ∈ [4, 5, 6, 7, …]).
- The numerator's coefficients argument of the filter (num) is a pointer on a real vector of **num_size** elements.
- The size argument (num_size) is the length of the numerator's coefficients of the filter (num_size∈ [1, 2, 3, …]).
- The denominator's coefficients argument of the filter (den) is a pointer on a real vector of **den_size** elements.
- The size argument (den_size) is the length of the denominator's coefficients of the filter (den_size∈ [1, 2, 3, …]).
- The predivisor (prediv) is used to scale down the denominator's coefficients of the filter in order to avoid overflow values. So when you use this feature, you have to prescale manually the denominator's coefficients by 2^prediv else leave this field to 0.

Algorithm

Following is the algorithm used to implement the IIR filter. The optimized version is based on this algorithm but can differ in certain points due to the instruction set of the target:

```
// Initialization of the vect1 coefficients
FOR n FROM 0 TO den_size - 1 DO
        sum1 = 0
        FOR m FROM 0 TO num_size - 1 DO
                sum1 += num[m] * vect2[n + num_size - m - 1]
        END

        sum2 = 0
        FOR m FROM 1 TO n DO
                sum2 += den[m] * vect1[n - m]
        END

        vect1[n] = (sum1 – (sum2 << prediv)) >> DSPXX_QB
END

FOR n FROM n TO size – num_size DO
        sum1 = 0
        FOR m FROM 0 TO num_size - 1 DO
                sum1 += num[m] * vect2[n + num_size - m - 1]
        END

        sum2 = 0
        FOR m FROM 1 TO den_size - 1 DO
                sum2 += den[m] * vect1[n - m]
        END

        vect1[n] = (sum1 – (sum2 << prediv)) >> DSPXX_QB
END
```

Notes

- **Interruptibility**: the code is interruptible.
- Due to its implementation, for the dsp16-avr32-uc3 optimized version of the FIR, the output buffer (vect1) has to have a length of 4*n elements to avoid overflows.
- The impulse response of the filter has to be scaled to avoid overflowing values.
- All the vectors have to be 32-bit aligned.
- The first denominator's coefficient have to be equal to 1 / (2^prediv).
- The predivisor (prediv) must be lower or equals to the constant DSPXX_QB.

Benchmark

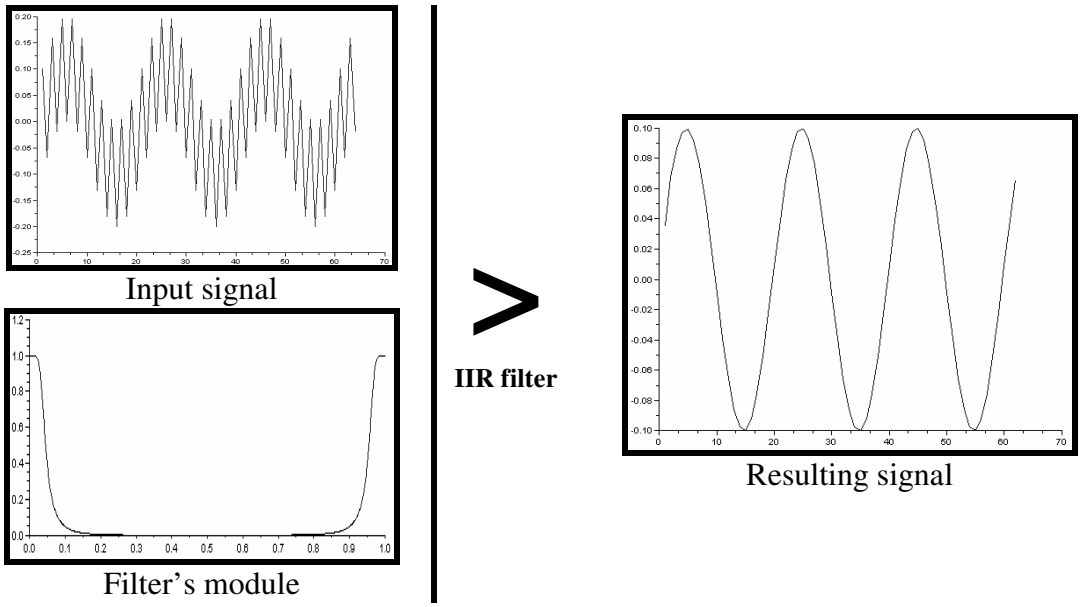
Benchmark routine

All these functions have been benchmarked on an avr32-uc3a0512 target. The programs have been compiled with avr32-gcc (4.1.2-atmel.1.0.0) with the –O3 optimization option and have been stored in FLASH memory. The fixed-point format used is the Q1.15 format for the 16-bit data and the Q1.31 format for the 32-bit data.

The benchmark process has been performed with the same input signal and impulse response for all those functions and compared with a reference's signal computed with a mathematic tool using floating point.

The input signal is a combination of one sine and one cosine. The sine oscillating at 400Hz and the cosine at 4KHz. Those signals have been added together and sampled at 8KHz.

The filter used is a low-pass Butterworth filter with a cutoff frequency equal to 2KHz.



Result

Here are tables of the main values of the benchmark results. All those values correspond to the best performances of the functions and are obtained with different compilation options. For more information, please refer to the complete benchmark result table in annexes.

16-bit IIR filter: generic

Concerned file path: */BASIC/FILTERING/dsp16_iir_generic.c*
Algorithm's size in memory: 266 bytes.
Order of the filter: 7

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
72-points	11,469	213.4us	1.40e-5	4.30e-5
256-points	44,591	829.8us	1.40e-5	4.30e-5
512-points	90,671	1.69ms	1.40e-5	4.30e-5
1024-points	182,831	3.40ms	1.40e-5	4.30e-5

More details on Table 4.1.1 in annexes

16-bit IIR filter: avr32-uc3 optimized

Concerned file path: */BASIC/FILTERING/dsp16_iir_avr32uc3.c*
Algorithm's size in memory: 1.0 Kbytes (size optimization), 3.1 Kbytes (speed optimization).
Order of the filter: 7

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
72-points	4,332	78.0us	1.90e-5	6.90e-5
256-points	15,006	270.5us	1.70e-5	6.90e-5
512-points	29,854	538.2us	1.70e-5	6.90e-5
1024-points	59,550	1.07ms	1.70e-5	6.90e-5

More details on Table 4.1.2 in annexes

32-bit IIR filter: generic

Concerned file path: */BASIC/FILTERING/dsp32_iir_generic.c*
Algorithm's size in memory: 400 bytes.
Order of the filter: 7

	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
72-points	14,517	265.4us	1.50e-9	7.00e-9
256-points	56,471	1.03ms	1.50e-9	7.00e-9
512-points	114,839	2.10ms	1.50e-9	7.00e-9
1024-points	231,575	4.23ms	1.50e-9	7.00e-9

More details on Table 4.2.1 in annexes

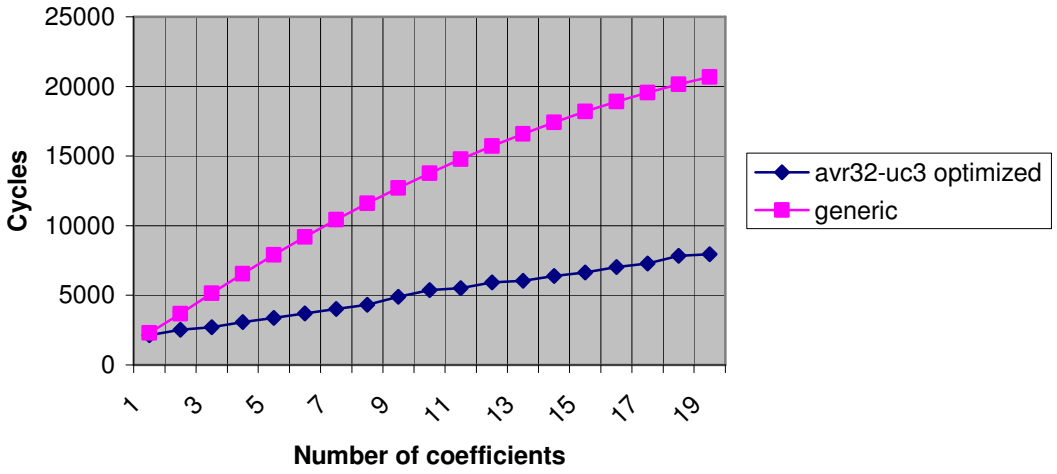
32-bit IIR filter: avr32-uc3 optimized

Concerned file path: */BASIC/FILTERING/dsp32_iir_avr32uc3.c*
Algorithm's size in memory: 3.0 Kbytes.
Order of the filter: 7

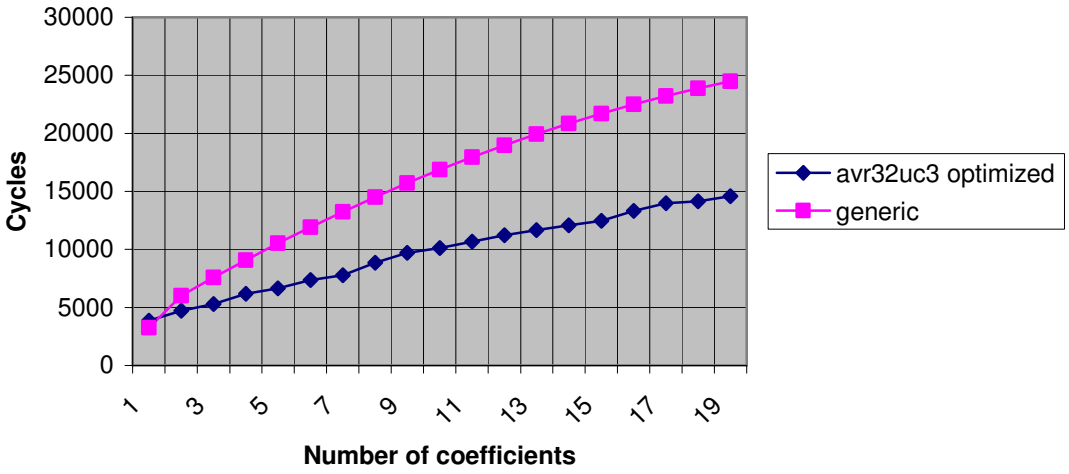
	Lowest cycle count	Fastest computation at 60 MHz	Lowest Error	
			Amplitude average	Max. amplitude
72-points	8,859	153.9us	1.50e-9	7.00e-9
256-points	33,333	577.1us	1.50e-9	7.00e-9
512-points	67,381	1.17ms	1.60e-9	7.00e-9
1024-points	135,477	2.34ms	1.60e-9	7.00e-9

More details on Table 4.2.2 in annexes

Benchmark results for the 16-bit version (with speed optimization)
The benchmark has been performed on a 72-element input signal.



Benchmark results for the 32-bit version (with speed optimization)
The benchmark has been performed on a 72-element input signal.



Remark: the number of coefficients corresponds to a filter which order is equal to "Number of coefficients" – 1.

Annexes

TABLE 1.1.1 - BENCHMARK OF 16-BIT RADIX-4 D.I.T. COMPLEX FFT: GENERIC

	Optimization	T.F.T.* in SRAM *(cycles)*		T.F.T.* in FLASH *(cycles)*		Error		Algorithm's size in memory *(bytes) (code size + T.F.T.* size)*
		wait-state		wait-state		*Amplitude average ($x10^{-5}$)*	*Max. amplitude ($x10^{-5}$)*	
		0 *at 30MHz*	**1** *at 60MHz*	**0** *at 30MHz*	**1** *at 60MHz*			
64-point	(0)	**6,296** (209.9us)	6,489 **(108.2us)**	6,640 (221.3us)	7,012 (116.9us)	2.98	15.63	1K *(840 + 194)*
	(1)	6,343 (211.4us)	6,538 (109.0us)	6,777 (225.9us)	7,167 (119.5us)	**1.58**	**6.53**	1K *(844 + 194)*
	(2)	6,666 (222.2us)	6,959 (116.0us)	7,092 (236.4us)	7,546 (125.8us)	2.98	15.63	**1.1K** *(1032 + 66)*
	(3)	6,747 (224.9us)	6,996 (116.6us)	7,101 (236.7us)	7,536 (125.6us)	1.58	6.53	1K *(984 + 66)*
256-point	(0)	**33,723** (1124.1us)	34,682 **(578.0us)**	35,265 (1175.5us)	37,033 (617.2us)	3.02	21.92	1.6K *(840 + 770)*
	(1)	34,041 (1134.7us)	35,003 (583.4us)	35,988 (1199.6us)	37,836 (630.6us)	**1.69**	**8.80**	1.6K *(844 + 770)*
	(2)	35,304 (1176.8us)	36,675 (611.3us)	37,194 (1239.8us)	39,294 (654.9us)	3.02	21.92	**1.3K** *(1032 + 258)*
	(3)	35,,826 (1194.2us)	37,032 (617.2us)	37,419 (1247.3us)	39,453 (657.6us)	1.69	8.80	1.2K *(984 + 258)*
1024-point	(0)	**169,006** (5.63ms)	173,611 **(2.90ms)**	175,394 (5.85ms)	183,358 (3.06ms)	3.04	25.01	3.8K *(840 + 3K)*
	(1)	170,795 (5.69ms)	175,404 (2.92ms)	178,863 (5.96ms)	187,161 (3.12ms)	**1.67**	**12.31**	3.8K *(844 + 3K)*
	(2)	175,446 (5.85ms)	181,703 (3.03ms)	183,248 (6.11ms)	192,530 (3.21ms)	3.04	25.01	**2K** *(1032 + 1K)*
	(3)	178,153 (5.94ms)	183,772 (3.06ms)	184,761 (6.16ms)	193,802 (3.23ms)	1.67	12.31	2K *(984 + 1K)*
4096-point	(0)	**812,321** (27.08ms)	833,820 **(13.90ms)**	838,147 (27.94ms)	873,235 (14.55ms)	3.09	32.90	12.8K *(840 + 12K)*
	(1)	821,533 (27.38ms)	843,037 (14.05ms)	854,154 (28.47ms)	890,598 (14.84ms)	**1.52**	**14.60**	12.8K *(844 + 12K)*
	(2)	838,212 (27.94ms)	866,315 (14.44ms)	869,718 (28.99ms)	910,054 (15.17ms)	3.09	32.90	**5K** *(1032 + 4K)*
	(3)	851,232 (28.37ms)	876,816 (14.61ms)	877,959 (29.27ms)	917,367 (15.29ms)	1.52	14.60	5K *(984 + 4K)*

*: Twiddle Factors Table.
(0): Algorithmic optimized for **speed.**
(1): Algorithmic optimized for **accuracy.**
(2): Algorithmic optimized for **size**.
(3): Algorithmic optimized for **size** and **accuracy**.

TABLE 1.1.2 - BENCHMARK OF 16-BIT RADIX-4 D.I.T. COMPLEX FFT: AVR32-UC3 OPTIMIZED

	Optimization	T.F.T.* in SRAM *(cycles)*		T.F.T.* in FLASH *(cycles)*		Error		Algorithm's size in memory *(bytes) (code size + T.F.T.* size)*
		wait-state		wait-state		*Amplitude average ($x10^{-5}$)*	*Max. amplitude ($x10^{-5}$)*	
		0 *at 30MHz*	**1** *at 60MHz*	**0** *at 30MHz*	**1** *at 60MHz*			
64-point	(0)	**2,611** (87.0us)	2,661 **(44.4us)**	2,753 (91.8us)	2,877 (48.0us)	3.00	10.04	894 *(700 + 194)*
	(1)	2,951 (98.4us)	2,999 (50.0us)	3,097 (103.2us)	3,265 (54.4us)	**1.63**	**6.53**	774 *(580 + 194)*
	(2)	2,833 (94.4us)	2,912 (48.5us)	3,027 (100.9us)	3,221 (53.7us)	2.93	10.04	**710** *(644 + 66)*
	(3)	3,206 (106.9us)	3,311 (55.2us)	3,458 (115.3us)	3,683 (61.4us)	1.63	6.53	766 *(700 + 66)*
256-point	(0)	**13,661** (455.4us)	13,932 **(232.2us)**	14,306 (476.9us)	14,904 (248.4us)	2.78	13.86	1.4K *(700 + 770)*
	(1)	15,777 (525.9us)	16,033 (267.2us)	16,428 (547.6us)	17,242 (287.4us)	**1.68**	**7.46**	1.3K *(580 + 770)*
	(2)	14,651 (488.4us)	15,056 (250.9us)	15,527 (517.6us)	16,442 (274.0us)	2.87	15.82	**902** *(644 + 258)*
	(3)	16,916 (563.9us)	17,469 (291.2us)	18,041 (601.4us)	19,152 (319.2us)	1.68	7.46	958 *(700 + 258)*
1024-point	(0)	**67,671** (2.26ms)	69,027 **(1.15ms)**	70,355 (2.35ms)	73,059 (1.22ms)	2.84	19.33	3.7K *(700 + 3K)*
	(1)	79,195 (2.64ms)	80,475 (1.34ms)	81,887 (2.73ms)	85,507 (1.43ms)	**1.69**	**10.23**	3.6K *(580 + 3K)*
	(2)	71,765 (2.39ms)	73,680 (1.23ms)	75,403 (2.51ms)	79,423 (1.32ms)	2.86	19.33	**1.6K** *(644 + 1K)*
	(3)	83,906 (2.80ms)	86,635 (1.44ms)	88,560 (2.95ms)	93,629 (1.56ms)	1.69	10.23	1.7K *(700 + 1K)*
4096-point	(0)	**322,897** (10.76ms)	329,370 **(5.49ms)**	333,764 (11.13ms)	345,678 (5.76ms)	2.80	23.69	12.7K *(700 + 12K)*
	(1)	381,269 (12.71ms)	387,413 (6.46ms)	392,146 (13.07ms)	407,788 (6.80ms)	**1.58**	**11.84**	12.6K *(580 + 12K)*
	(2)	339,439 (11.31ms)	348,176 (5.80ms)	354,159 (11.81ms)	371,396 (6.19ms)	2.81	23.69	**4.6K** *(644 + 4K)*
	(3)	400,304 (13.34ms)	413,273 (6.89ms)	419,111 (13.97ms)	441,578 (7.36ms)	1.58	11.84	4.7K *(700 + 4K)*

*: Twiddle Factors Table.
(0): Algorithmic optimized for **speed.**
(1): Algorithmic optimized for **accuracy.**
(2): Algorithmic optimized for **size.**
(3): Algorithmic optimized for **size** and **accuracy.**

TABLE 1.2.1 - BENCHMARK OF 32-BIT RADIX-4 D.I.T. COMPLEX FFT: GENERIC

	Optimization	T.F.T.* in SRAM *(cycles)*		T.F.T.* in FLASH *(cycles)*		Error		Algorithm's size in memory *(bytes) (code size + T.F.T.* size)*
		wait-state		wait-state		*Amplitude average ($x10^{-9}$)*	*Max. amplitude ($x10^{-9}$)*	
		0 *at 30MHz*	**1** *at 60MHz*	**0** *at 30MHz*	**1** *at 60MHz*			
64-point	(0)	**13,206** (440.2us)	13,509 **(225.2us)**	13,580 (452.7us)	14,063 (234.4us)	0.70	5.70	2.2K *(1816 + 392)*
	(1)	15,323 (510.8us)	15,659 (261.0us)	15,627 (520.9us)	16,113 (268.6us)	**0.60**	**5.70**	2.4K *(2112 + 392)*
	(2)	13,622 (454.1us)	14,011 (233.5us)	13,938 (464.6us)	14,497 (241.6us)	0.70	5.70	**2.0K** *(1952 + 136)*
	(3)	15,714 (523.8us)	16,245 (270.8us)	16,058 (535.3us)	16,805 (280.1us)	0.60	5.70	2.3K *(2248 + 136)*
256-point	(0)	**74,297** (2.48ms)	75,940 **(1.27ms)**	75,992 (2.53ms)	78,445 (1.31ms)	0.50	4.80	3.3K *(1816 + 1.5K)*
	(1)	87,791 (2.93ms)	89,605 (1.49ms)	89,165 (2.97ms)	91,636 (1.53ms)	**0.30**	**4.80**	3.6K *(2112 + 1.5K)*
	(2)	76,136 (2.54ms)	78,160 (1.30ms)	77,537 (2.58ms)	80,329 (1.34ms)	0.50	4.80	**2.4K** *(1952 + 520)*
	(3)	89,543 (2.98ms)	92,446 (1.54ms)	91,058 (3.04ms)	94,978 (1.59ms)	0.30	4.80	2.7K *(2248 + 520)*
1024-point	(0)	**383,212** (12.77ms)	391,715 **(6.53ms)**	390,260 (13.01ms)	402,123 (6.70ms)	0.50	8.80	7.8K *(1816 + 6K)*
	(1)	457,571 (15.25ms)	466,815 (7.78ms)	463,279 (15.44ms)	475,223 (7.92ms)	**0.30**	**6.10**	8.1K *(2112 + 6K)*
	(2)	390,794 (13.03ms)	400,869 (6.68ms)	396,576 (13.22ms)	409,841 (6.83ms)	0.50	8.80	**3.9K** *(1952 + 2K)*
	(3)	464,988 (15.50ms)	479,847 (8.00ms)	471,226 (15.71ms)	490,367 (8.17ms)	0.30	6.10	4.2K *(2248 + 2K)*

*: Twiddle Factors Table.
(0): Algorithmic optimized for **speed.**
(1): Algorithmic optimized for **accuracy.**
(2): Algorithmic optimized for **size.**
(3): Algorithmic optimized for **size** and **accuracy.**

TABLE 2.1.1 - BENCHMARK OF 16-BIT CONVOLUTION: GENERIC

1st input signal	2nd input signal	Execution time (cycles)		Error	
		wait-state		*Amplitude average* ($x10^{-5}$)	*Max. amplitude* ($x10^{-5}$)
		0 *at 30MHz*	**1** *at 60MHz*		
32-point	**32-points**	**15,681** (522.7us)	16,344 (**272.4us**)	1.60	3.90
	64-points	**23,524** (784.1us)	24,508 (**408.5us**)	2.00	4.50
	128-points	**39,204** (1.31ms)	40,830 (**680.5us**)	1.90	4.50
	256-points	**70,564** (2.35ms)	73,470 (**1.22ms**)	1.70	4.10
64-point	**64-points**	**57,757** (1.93ms)	60,084 (**1.00ms**)	1.80	4.70
	128-points	**86,752** (2.89ms)	90,234 (**1.50ms**)	1.80	4.40
	256-points	**144,736** (4.82ms)	150,522 (**2.51ms**)	1.50	4.80
128-point	**128-points**	**221,780** (7.39ms)	230,512 (**3.84ms**)	1.80	5.30
	256-points	**333,018** (11.10ms)	346,097 (**5.77ms**)	1.70	5.00
256-point	**256-points**	**869,316** (28.98ms)	903,136 (**15.05ms**)	1.70	5.40

TABLE 2.1.2 - BENCHMARK OF 16-BIT CONVOLUTION: AVR32-UC3 OPTIMIZED

1st input signal	2nd input signal	Execution time (cycles)		Error	
		wait-state		*Amplitude average* ($x10^{-5}$)	*Max. amplitude* ($x10^{-5}$)
		0 *at 30MHz*	**1** *at 60MHz*		
32-point	**32-points**	**5,571** (185.7us)	6,127 (**102.1us**)	1.60	3.90
	64-points	**8,248** (274.9us)	9,070 (**151.2us**)	2.00	4.50
	128-points	**13,592** (453.1us)	14,944 (**249.1us**)	1.90	4.50
	256-points	**24,280** (809.3us)	26,688 (**444.8us**)	1.70	4.10
64-point	**64-points**	**19,087** (636.2us)	20,947 (**349.1us**)	1.80	4.70
	128-points	**28,532** (951.1us)	31,308 (**521.8us**)	1.80	4.40
	256-points	**47,412** (1.58ms)	52,012 (**866.9us**)	1.50	4.80
128-point	**128-points**	**70,694** (2.36ms)	77,471 (**1.29ms**)	1.80	5.30
	256-points	**105,966** (3.53ms)	116,099 (**1.93ms**)	1.70	5.00
256-point	**256-points**	**272,214** (9.07ms)	298,031 (**4.97ms**)	1.70	5.40

TABLE 2.2.1 - BENCHMARK OF 32-BIT CONVOLUTION: GENERIC

1st input signal	2nd input signal	Execution time (cycles)		Error	
		wait-state		*Amplitude average* ($x10^{-9}$)	*Max. amplitude* ($x10^{-9}$)
		0 *at 30MHz*	**1** *at 60MHz*		
32-point	**32-points**	**28,359** (945.3us)	29,182 (**486.4us**)	0.40	1.60
	64-points	**42,572** (1.42ms)	43,788 (**729.8us**)	0.40	2.10
	128-points	**70,988** (2.37ms)	72,990 (**1.22ms**)	0.30	1.90
	256-points	**127,820** (4.26ms)	131,390 (**2.19ms**)	0.40	1.70
64-point	**64-points**	**109,179** (3.64ms)	112,334 (**1.87ms**)	0.40	1.70
	128-points	**163,968** (5.47ms)	168,678 (**2.81ms**)	0.50	1.60
	256-points	**273,536** (9.12ms)	281,350 (**4.69ms**)	0.60	2.70
128-point	**128-points**	**429,026** (14.30ms)	441,458 (**7.36ms**)	0.40	2.30
	256-points	**644,074** (21.47ms)	662,677 (**11.04ms**)	0.40	2.00
256-point	**256-points**	**1,701,554** (56.72ms)	1,750,962 (**29.18ms**)	0.50	3.10

TABLE 2.2.2 - BENCHMARK OF 32-BIT CONVOLUTION: AVR32-UC3 OPTIMIZED

1st input signal	2nd input signal	Execution time (cycles)		Error	
		wait-state		*Amplitude average* ($x10^{-9}$)	*Max. amplitude* ($x10^{-9}$)
		0 *at 30MHz*	**1** *at 60MHz*		
32-point	**32-points**	**13,361** (445.4us)	13,680 (**228.0us**)	0.60	2.30
	64-points	**19,958** (665.3us)	20,414 (**340.2us**)	0.50	2.10
	128-points	**33,142** (1.10ms)	33,872 (**564.5us**)	0.50	1.90
	256-points	**59,510** (1.98ms)	60,784 (**1.01ms**)	0.50	2.10
64-point	**64-points**	**50,501** (1.68ms)	51,624 (**860.4us**)	0.50	1.70
	128-points	**75,722** (2.52ms)	77,376 (**1.29ms**)	0.60	2.40
	256-points	**126,154** (4.21ms)	128,864 (**2.15ms**)	0.70	2.70
128-point	**128-points**	**196,972** (6.57ms)	201,244 (**3.35ms**)	0.50	2.30
	256-points	**295,540** (9.85ms)	301,887 (**5.03ms**)	0.60	2.30
256-point	**256-points**	**778,684** (25.96ms)	795,388 (**13.26ms**)	0.60	3.10

TABLE 3.1.1 - BENCHMARK OF 16-BIT FIR FILTER: GENERIC

	Number of Taps	I.R.* in SRAM *(cycles)*		I.R.* in FLASH *(cycles)*		Error	
		wait-state		wait-state		*Amplitude average ($x10^{-5}$)*	*Max. amplitude ($x10^{-5}$)*
		0 *at 30MHz*	**1** *at 60MHz*	**0** *at 30MHz*	**1** *at 60MHz*		
64-point	24	**7,424** (247.5us)	7,682 (**128.0us**)	10,704 (356.8us)	12,352 (205.9us)	2.27	9.46
	48	**5,780** (192.7us)	5,996 (**99.9us**)	8,517 (283.9us)	9,868 (164.5us)	3.09	15.57
256-point	24	**41,793** (1.39ms)	43,202 (**720.0us**)	60,433 (2.01ms)	69,760 (1.16ms)	2.22	9.46
	48	**70,101** (2.34ms)	72,620 (**1.21ms**)	103,750 (3.46ms)	120,268 (2.00ms)	5.66	27.20
	72	**90,921** (3.03ms)	94,262 (**1.57ms**)	135,691 (4.52ms)	157,528 (2.63ms)	11.25	52.65
	100	**105,127** (3.50ms)	108,903 (**1.82ms**)	156,466 (5.22ms)	185,989 (3.10ms)	14.19	63.71
512-point	24	**87,617** (2.92ms)	90,562 (**1.51ms**)	126,737 (4.22ms)	146,304 (2.44ms)	2.23	9.46
	48	**155,861** (5.20ms)	161,452 (**2.69ms**)	230,726 (7.69ms)	267,468 (4.46ms)	5.78	27.20
	72	**216,617** (7.22ms)	224,566 (**3.74ms**)	323,339 (10.78ms)	375,384 (6.26ms)	10.79	52.65
	100	**276,391** (9.21ms)	286,311 (**4.77ms**)	411,442 (13.71ms)	489,093 (8.15ms)	14.05	63.71
1024-point	24	**179,265** (5.98ms)	185,282 (**3.09ms**)	259,345 (8.64ms)	299,392 (4.99ms)	2.21	9.46
	48	**327,381** (10.91ms)	339,116 (**5.65ms**)	484,678 (16.16ms)	561,868 (9.36ms)	5.85	27.20
	72	**468,009** (15.60ms)	485,174 (**8.09ms**)	698,635 (23.29ms)	811,096 (13.52ms)	10.69	52.65
	100	**618,919** (20.63ms)	641,127 (**10.69ms**)	921,394 (30.71ms)	1,095,301 (18.26ms)	13.99	63.71

*: Impulse Response.

TABLE 3.1.2 - BENCHMARK OF 16-BIT FIR FILTER: AVR32-UC3 OPTIMIZED

	Number of Taps	I.R.* in SRAM *(cycles)*		I.R.* in FLASH *(cycles)*		Error	
		wait-state		wait-state		*Amplitude average ($x10^{-5}$)*	*Max. amplitude ($x10^{-5}$)*
		0 *at 30MHz*	**1** *at 60MHz*	**0** *at 30MHz*	**1** *at 60MHz*		
64-point	24	**2,439** (81.3us)	2,657 (**44.3us**)	2,703 (90.1us)	3,054 (50.9us)	2.27	9.46
	48	**2,115** (70.5us)	2,309 (**38.5us**)	2,355 (78.5us)	2,670 (44.5us)	3.09	15.57
256-point	24	**12,712** (423.7us)	13,841 (**230.7us**)	14,128 (470.9us)	15,966 (266.1us)	2.22	9.46
	48	**21,604** (720.1us)	23,573 (**392.9us**)	24,148 (804.9us)	27,390 (456.5us)	5.66	27.20
	72	**28,192** (939.7us)	30,785 (**513.1us**)	31,576 (1.05ms)	35,862 (597.7us)	11.25	52.65
	100	**32,966** (1.10ms)	36,014 (**600.2us**)	36,966 (1.23ms)	42,015 (700.3us)	14.19	63.71
512-point	24	**26,408** (880.3us)	28,753 (**479.2us**)	29,360 (978.7us)	33,182 (553.0us)	2.23	9.46
	48	**47,588** (1.59ms)	51,925 (**865.4us**)	53,204 (1.77ms)	60,350 (1.01ms)	5.78	27.20
	72	**66,464** (2.22ms)	72,577 (**1.21ms**)	74,456 (2.48ms)	84,566 (1.41ms)	10.79	52.65
	100	**85,574** (2.85ms)	93,486 (**1.56ms**)	95,974 (3.20ms)	109,087 (1.82ms)	14.05	63.71
1024-point	24	**53,800** (1.79ms)	58,577 (**976.3us**)	59,824 (1.99ms)	67,614 (1.13ms)	2.21	9.46
	48	**99,556** (3.32ms)	108,629 (**1.81ms**)	111,316 (3.71ms)	126,270 (2.10ms)	5.85	27.20
	72	**143,008** (4.77ms)	156,161 (**2.60ms**)	160,216 (5.34ms)	181,974 (3.03ms)	10.69	52.65
	100	**190,790** (6.36ms)	208,430 (**3.47ms**)	213,990 (7.13ms)	243,231 (4.05ms)	13.99	63.71

*: Impulse Response.

TABLE 3.2.1 - BENCHMARK OF 32-BIT FIR FILTER: GENERIC

	Number of Taps	I.R.* in SRAM *(cycles)*		I.R.* in FLASH *(cycles)*		Error	
		wait-state		wait-state		*Amplitude average ($x10^{-9}$)*	*Max. amplitude ($x10^{-9}$)*
		0 *at 30MHz*	**1** *at 60MHz*	**0** *at 30MHz*	**1** *at 60MHz*		
64-point	24	**13,984** (466.1us)	14,365 (**239.4us**)	16,608 (553.6us)	18,624 (310.4us)	2.10	12.40
	48	**11,101** (370.0us)	11,419 (**190.3us**)	13,311 (443.7us)	14,865 (247.8us)	2.50	14.40
256-point	24	**79,073** (2.64ms)	81,181 (**1.35ms**)	93,985 (3.13ms)	105,408 (1.76ms)	2.30	17.40
	48	**135,518** (4.52ms)	139,291 (**2.32ms**)	162,688 (5.42ms)	181,713 (3.03ms)	2.70	16.60
	72	**177,131** (5.90ms)	182,137 (**3.04ms**)	213,391 (7.11ms)	238,002 (3.97ms)	4.60	31.50
	100	**187,238** (6.24ms)	194,313 (**3.24ms**)	241,717 (8.06ms)	264,808 (4.41ms)	4.60	31.90
512-point	24	**165,857** (5.53ms)	170,269 (**2.84ms**)	197,153 (6.57ms)	221,120 (3.69ms)	2.60	23.10
	48	**301,406** (10.05ms)	309,787 (**5.16ms**)	361,856 (12.06ms)	404,177 (6.74ms)	3.20	23.90
	72	**422,123** (14.07ms)	434,041 (**7.23ms**)	508,559 (16.95ms)	567,218 (9.45ms)	5.10	35.30
	100	**492,390** (16.41ms)	510,985 (**8.52ms**)	635,701 (21.19ms)	696,424 (11.61ms)	5.10	31.90
1024-point	24	**339,425** (11.31ms)	348,445 (**5.81ms**)	403,489 (13.45ms)	452,544 (7.54ms)	3.70	28.40
	48	**633,182** (21.11ms)	650,779 (**10.85ms**)	760,192 (25.34ms)	849,105 (14.15ms)	4.60	29.30
	72	**912,107** (30.40ms)	937,849 (**15.63ms**)	1,098,895 (36.63ms)	1,225,650 (20.43ms)	6.50	35.30
	100	**1,102,694** (36.76ms)	1,144,329 (**19.07ms**)	1,423,669 (47.46ms)	1,559,656 (26.00ms)	6.40	33.30

*: Impulse Response.

TABLE 3.2.2 - BENCHMARK OF 32-BIT FIR FILTER: AVR32-UC3 OPTIMIZED

	Number of Taps	I.R.* in SRAM *(cycles)*		I.R.* in FLASH *(cycles)*		Error	
		wait-state		wait-state		*Amplitude average ($x10^{-9}$)*	*Max. amplitude ($x10^{-9}$)*
		0 *at 30MHz*	**1** *at 60MHz*	**0** *at 30MHz*	**1** *at 60MHz*		
64-point	24	**6,479** (216.0us)	6,613 (**110.2us**)	9,141 (304.7us)	10,918 (182.0us)	2.10	12.40
	48	**5,132** (171.1us)	5,245 (**87.4us**)	7,305 (243.5us)	8,866 (147.8us)	2.50	14.40
256-point	24	**36,432** (1.21ms)	37,140 (**619.0us**)	51,574 (1.72ms)	61,605 (1.03ms)	2.30	12.40
	48	**62,157** (2.07ms)	63,420 (**1.06ms**)	88,906 (2.96ms)	107,937 (1.80ms)	2.70	16.60
	72	**81,114** (2.70ms)	82,788 (**1.38ms**)	116,446 (3.88ms)	142,173 (2.37ms)	4.60	31.50
	100	**94,441** (3.15ms)	96,490 (**1.61ms**)	140,910 (4.70ms)	166,671 (2.78ms)	4.60	31.90
512-point	24	**76,368** (2.55ms)	77,844 (**1.30ms**)	108,150 (3.61ms)	129,189 (2.15ms)	2.60	23.10
	48	**138,189** (4.61ms)	140,988 (**2.35ms**)	197,706 (6.59ms)	240,033 (4.00ms)	3.20	23.90
	72	**193,242** (6.44ms)	197,220 (**3.29ms**)	277,470 (9.25ms)	338,781 (5.65ms)	5.10	35.30
	100	**248,297** (8.28ms)	253,674 (**4.23ms**)	370,542 (12.35ms)	438,287 (7.30ms)	5.10	31.90
1024-point	24	**156,240** (5.21ms)	159,252 (**2.65ms**)	221,302 (7.38ms)	264,357 (4.41ms)	3.70	28.40
	48	**290,253** (9.68ms)	296,124 (**4.94ms**)	415,306 (13.84ms)	504,225 (8.40ms)	4.50	29.30
	72	**417,498** (13.92ms)	426,084 (**7.10ms**)	599,518 (19.99ms)	731,997 (12.20ms)	6.50	35.30
	100	**556,009** (18.53ms)	568,042 (**9.47ms**)	829,806 (27.66ms)	981,519 (16.36ms)	6.40	33.30

*: Impulse Response.

TABLE 4.1.1 - BENCHMARK OF 16-BIT FIR FILTER: GENERIC

	Order of the filter	Execution time *(cycles)*		Error	
		wait-state		*Amplitude average ($x10^{-5}$)*	*Max. amplitude ($x10^{-5}$)*
		0 *at 30MHz*	**1** *at 60MHz*		
72-point	2	**5,294** (176.5us)	5,717 (**95.3us**)	1.50	4.00
	7	**11,469** (382.3us)	12,802 (**213.4us**)	1.40	4.30
	12	**16,319** (544.0us)	18,387 (**306.5us**)	2.50	7.60
256-point	2	**19,096** (636.5us)	20,621 (**343.7us**)	1.50	4.00
	7	**44,591** (1.49ms)	49,786 (**829.8us**)	1.40	4.30
	12	**68,761** (2.29ms)	77,451 (**1.29ms**)	3.50	8.90
512-point	2	**38,296** (1.28ms)	41,357 (**689.3us**)	1.50	4.00
	7	**90,671** (3.02ms)	101,242 (**1.69ms**)	1.40	4.30
	12	**141,721** (4.72ms)	159,627 (**2.66ms**)	3.70	8.90
1024-point	2	**76,696** (2.56ms)	82,829 (**1.38ms**)	1.50	4.00
	7	**182,831** (6.09ms)	204,154 (**3.40ms**)	1.40	4.30
	12	**287,641** (9.59ms)	323,979 (**5.40ms**)	3.90	8.90

TABLE 4.1.2 - BENCHMARK OF 16-BIT FIR FILTER: AVR32-UC3 OPTIMIZED

	Order of the filter	Execution time *(cycles)*		Error	
		wait-state		*Amplitude average ($x10^{-5}$)*	*Max. amplitude ($x10^{-5}$)*
		0 *at 30MHz*	**1** *at 60MHz*		
72-point	2	**2,725** (90.8us)	2,871 (**47.9us**)	1.20	4.00
	7	**4,332** (144.4us)	4,682 (**78.0us**)	1.90	6.90
	12	**6,089** (203.0us)	6,534 (**108.9us**)	3.30	8.40
256-point	2	**9,167** (305.6us)	9,633 (**160.6us**)	1.10	4.00
	7	**15,006** (500.2us)	16,228 (**270.5us**)	1.70	6.90
	12	**22,283** (742.8us)	23,876 (**397.9us**)	5.10	12.40
512-point	2	**18,127** (604.2us)	19,041 (**317.4us**)	1.10	4.00
	7	**29,854** (995.1us)	32,292 (**538.2us**)	1.70	6.90
	12	**44,811** (1.49ms)	48,004 (**800.1us**)	5.60	12.40
1024-point	2	**36,047** (1.20ms)	37,857 (**631.0us**)	1.10	4.00
	7	**59,550** (1.99ms)	64,420 (**1.07ms**)	1.70	6.90
	12	**89,867** (3.00ms)	96,260 (**1.60ms**)	5.80	12.40

TABLE 4.2.2 - BENCHMARK OF 32-BIT FIR FILTER: GENERIC

	Order of the filter	Execution time *(cycles)*		Error	
		wait-state		*Amplitude average ($x10^{-9}$)*	*Max. amplitude ($x10^{-9}$)*
		0 *at 30MHz*	**1** *at 60MHz*		
72-point	2	**7,582** (252.7us)	8,072 (**134.5us**)	1.00	3.80
	7	**14,517** (483.9us)	15,922 (**265.4us**)	1.50	7.00
	12	**19,952** (665.1us)	22,097 (**368.3us**)	2.60	10.30
256-point	2	**27,456** (915.2us)	29,232 (**487.2us**)	0.80	3.80
	7	**56,471** (1.88ms)	61,922 (**1.03ms**)	1.50	7.00
	12	**83,986** (2.80ms)	92,937 (**1.55ms**)	1.50	10.30
512-point	2	**55,104** (1.84ms)	58,672 (**977.9us**)	0.80	3.80
	7	**114,839** (3.83ms)	125,922 (**2.10ms**)	1.50	7.00
	12	**173,074** (5.77ms)	191,497 (**3.19ms**)	1.40	10.30
1024-point	2	**110,400** (3.68ms)	117,552 (**1.96ms**)	0.80	3.80
	7	**231,575** (7.72ms)	253,922 (**4.23ms**)	1.50	7.00
	12	**351,250** (11.71ms)	388,617 (**6.48ms**)	1.30	10.30

TABLE 4.2.1 - BENCHMARK OF 32-BIT FIR FILTER: AVR32-UC3 OPTIMIZED

	Order of the filter	Execution time *(cycles)*		Error	
		wait-state		*Amplitude average ($x10^{-9}$)*	*Max. amplitude ($x10^{-9}$)*
		0 *at 30MHz*	**1** *at 60MHz*		
72-point	2	**5,297** (176.6us)	5,662 (**94.4us**)	1.00	3.80
	7	**8,859** (295.3us)	9,233 (**153.9us**)	1.50	7.00
	12	**11,670** (389.0us)	12,239 (**204.0us**)	3.00	10.30
256-point	2	**18,731** (624.4us)	20,014 (**333.6us**)	0.80	3.80
	7	**33,333** (1.11ms)	34,625 (**577.1us**)	1.50	7.00
	12	**46,816** (1.56ms)	48,855 (**814.3us**)	1.80	10.30
512-point	2	**37,419** (1.25ms)	39,982 (**666.4us**)	0.80	3.80
	7	**67,381** (2.25ms)	69,953 (**1.17ms**)	1.60	7.00
	12	**95,712** (3.19ms)	99,799 (**1.66ms**)	1.70	10.30
1024-point	2	**74,795** (2.49ms)	79,918 (**1.33ms**)	0.80	3.80
	7	**135,477** (4.52ms)	140,609 (**2.34ms**)	1.60	7.00
	12	**193,504** (6.45ms)	201,687 (**3.36ms**)	1.60	10.30

TI Bench results

TABLE OF CONTENTS

/!\ The concurrency used IAR to perform this benchmark. Therefore, the results obtained with GCC for the Whetstone and the Dhrystone are not comparable. This is due to the suppression of useless sections in the code in full optimization mode.

AVR32-UC3 "TI BENCH" results

	GCC		IAR	
	No opti.	*Full opti.*	*No opti.*	*Full opti.*
8-bit 2-dim Matrix	3207	611	2120	867
8-bit Math	206	73	101	33
8-bit Switch Case	82	17	67	60
16-bit 2-dim Matrix	3098	611	2090	867
16-bit Math	203	76	102	33
16-bit Switch Case	82	17	67	60
32-bit Math	186	73	99	33
Dhrystone	116360	25283 !	114285	52789
FIR Filter	128781	109559	72491	69175
Floating-point Math	850	717	613	573
Matrix Multiplication	3838	621	2238	939
Whetstone	141067	59424 !	144736	143502

Global comparison with several microcontrollers

AVR32-UC3A – GCC Compiler

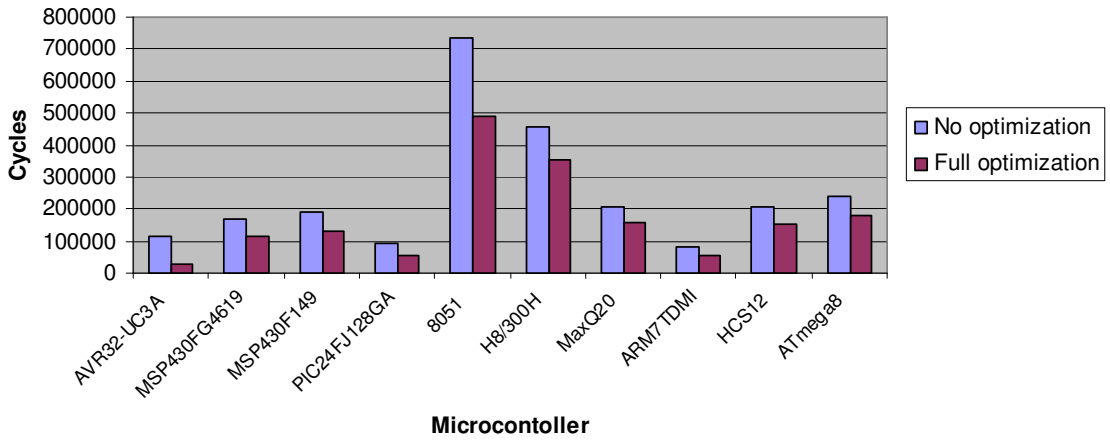
Cycle Count without Optimization for Dhrystone

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Dhrystone	116360	166457	190834	93920	***	732532	454518	207905	83798	208648	240320

Cycle Count with Optimization for Dhrystone

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Dhrystone	25283 !	111929	130104	52081	***	488193	352510	157965	52352	152212	179834

Cycle Count for Dhrystone



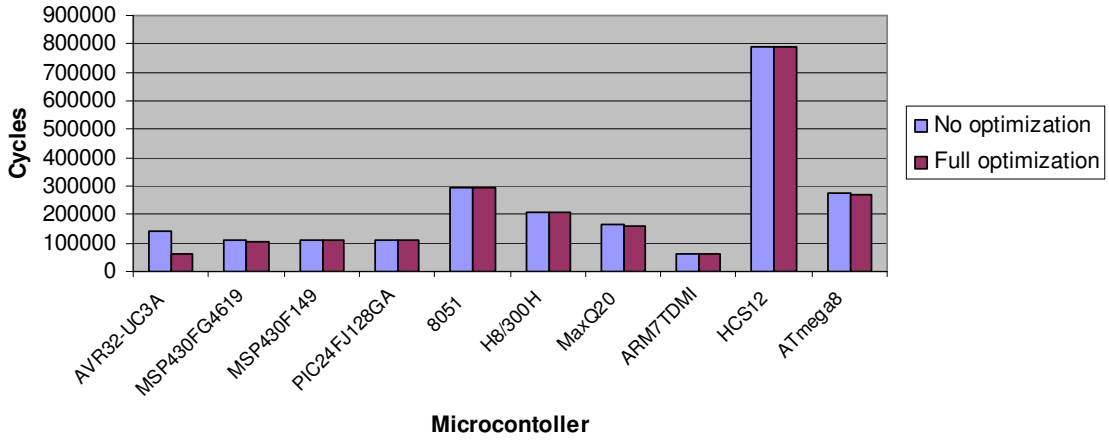
Cycle Count without Optimization for Whetstone

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Whetstone	141067	108093	111102	109807	***	294309	209370	162541	61600	788966	274586

Cycle Count with Optimization for Whetstone

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Whetstone	59424 !	107040	109837	108619	***	291836	205910	158945	60444	787635	270991

Cycle Count for Whetstone



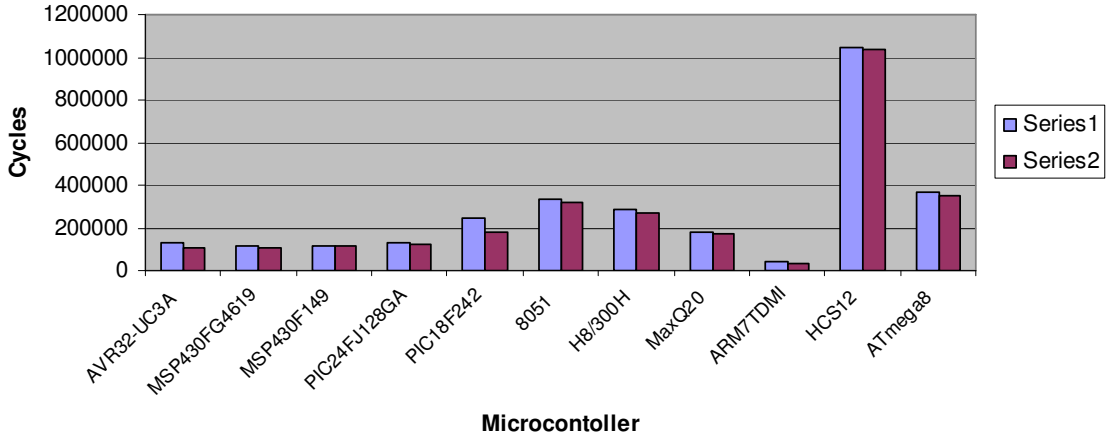
Cycle Count without Optimization for FIR Filter

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Whetstone	128781	113308	118170	127125	245704	330640	285580	176720	37827	1045982	365837

Cycle Count with Optimization for FIR Filter

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Whetstone	109559	108527	113399	122692	182210	321781	271964	167583	33114	1035394	352894

Cycle Count for FIR Filter Operation



Cycle Count without Optimization for Simple Math Operations

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
8-Bit Math	206	250	261	107	141	212	240	175	87	97	134
8-Bit Matrix	3207	2370	2497	2927	7310	14898	10228	6196	2122	6858	2523
8-Bit Switch	82	33	32	76	49	112	96	42	51	51	39
16-Bit Math	203	223	233	108	332	542	254	201	102	108	288

16-Bit Matrix	3098	3140	3270	3183	26533	23868	11252	9012	2890	8650	9506
16-Bit Switch	82	32	31	74	87	314	102	35	51	54	45
32-Bit Math	186	569	589	564	1259	3854	520	440	109	267	750
Floating-Point Math	850	771	795	789	1049	3339	1548	644	205	5508	1663
Matrix Multiplication	3838	4500	4706	3203	32096	19856	14018	9624	3424	8034	8417
Total	11752	11888	12414	11031	68856	66995	38258	26369	9041	29627	23365

Cycle Count with Full Optimization for Simple Math Operations

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
8-Bit Math	73	233	243	75	136	176	152	130	64	68	110
8-Bit Matrix	611	875	1009	1051	2193	2590	4362	1140	475	1559	984
8-Bit Switch	17	32	31	61	49	112	62	38	20	46	38
16-Bit Math	76	210	219	73	339	526	172	183	79	60	266
16-Bit Matrix	611	811	945	1115	6461	4294	4746	1508	475	2073	1488
16-Bit Switch	17	31	30	60	87	318	66	34	20	41	44
32-Bit Math	73	556	575	510	1284	2622	388	425	97	235	731
Floating-Point Math	717	762	786	741	1085	2127	1416	629	187	5470	1654
Matrix Multiplication	621	2550	2762	1384	5283	5880	10468	2214	839	2732	2396
Total	2816	6060	6600	5070	16917	18645	21832	6301	2256	12284	7711

AVR32-UC3A – IAR Compiler

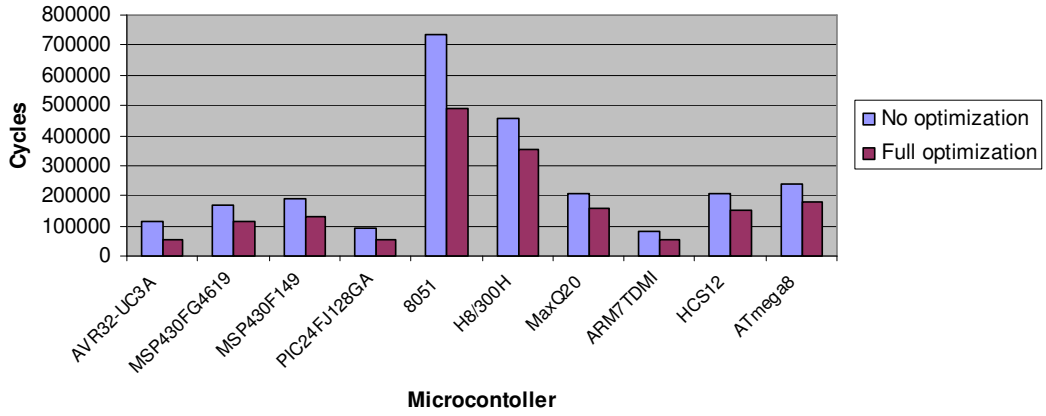
Cycle Count without Optimization for Dhrystone

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Dhrystone	114285	166457	190834	93920	***	732532	454518	207905	83798	208648	240320

Cycle Count with Optimization for Dhrystone

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Dhrystone	52789	111929	130104	52081	***	488193	352510	157965	52352	152212	179834

Cycle Count for Dhrystone



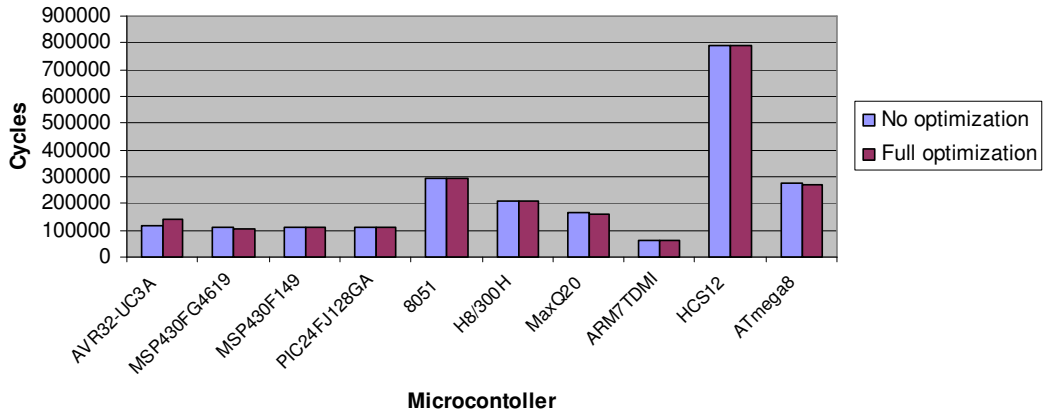
Cycle Count without Optimization for Whetstone

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Whetstone	114736	108093	111102	109807	***	294309	209370	162541	61600	788966	274586

Cycle Count with Optimization for Whetstone

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
Whetstone	143502	107040	109837	108619	***	291836	205910	158945	60444	787635	270991

Cycle Count for Whetstone



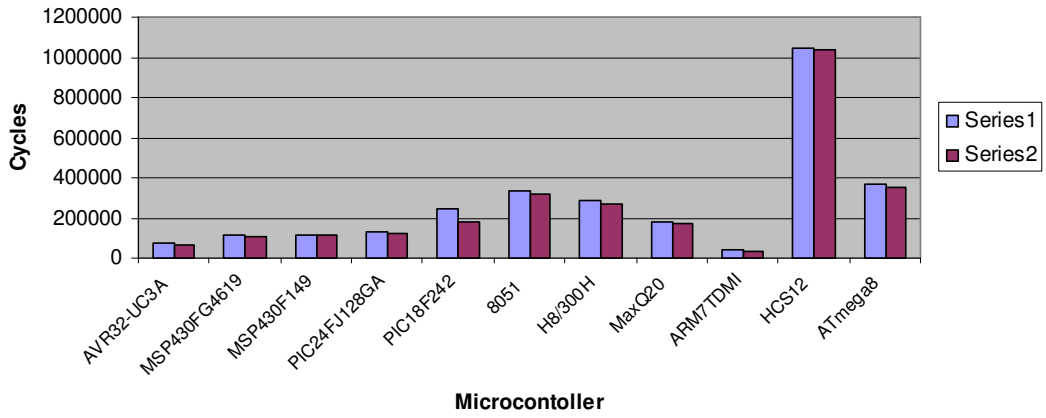
Cycle Count without Optimization for FIR Filter

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
FIR Filter	72491	113308	118170	127125	245704	330640	285580	176720	37827	1045982	365837

Cycle Count with Optimization for FIR Filter

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
FIR Filter	69175	108527	113399	122692	182210	321781	271964	167583	33114	1035394	352894

Cycle Count for FIR Filter Operation



Cycle Count without Optimization for Simple Math Operations

	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
8-Bit Math	101	250	261	107	141	212	240	175	87	97	134
8-Bit Matrix	2120	2370	2497	2927	7310	14898	10228	6196	2122	6858	2523
8-Bit Switch	67	33	32	76	49	112	96	42	51	51	39
16-Bit Math	102	223	233	108	332	542	254	201	102	108	288
16-Bit Matrix	2090	3140	3270	3183	26533	23868	11252	9012	2890	8650	9506
16-Bit Switch	67	32	31	74	87	314	102	35	51	54	45
32-Bit Math	99	569	589	564	1259	3854	520	440	109	267	750

Floating-Point Math	613	771	795	789	1049	3339	1548	644	205	5508	1663
Matrix Multiplication	2238	4500	4706	3203	32096	19856	14018	9624	3424	8034	8417
Total	7497	11888	12414	11031	68856	66995	38258	26369	9041	29627	23365

Cycle Count with Full Optimization for Simple Math Operations

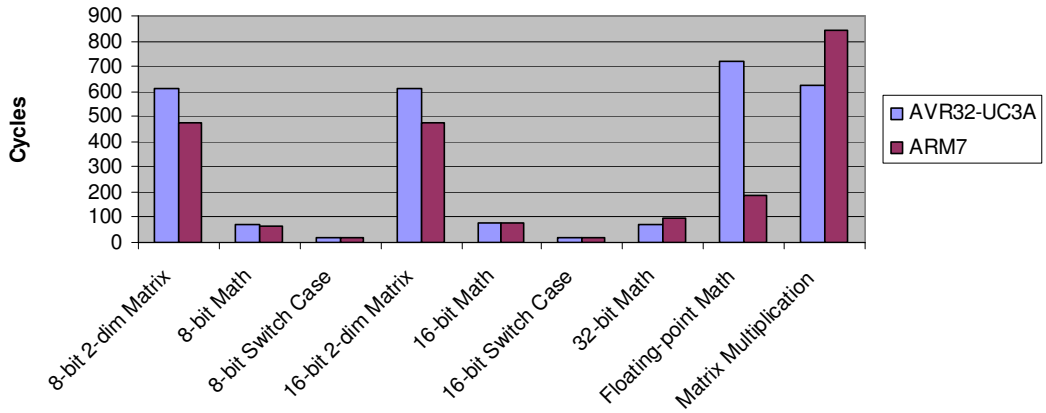
	AVR32-UC3A	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
8-Bit Math	33	233	243	75	136	176	152	130	64	68	110
8-Bit Matrix	867	875	1009	1051	2193	2590	4362	1140	475	1559	984
8-Bit Switch	60	32	31	61	49	112	62	38	20	46	38
16-Bit Math	33	210	219	73	339	526	172	183	79	60	266
16-Bit Matrix	867	811	945	1115	6461	4294	4746	1508	475	2073	1488
16-Bit Switch	60	31	30	60	87	318	66	34	20	41	44
32-Bit Math	33	556	575	510	1284	2622	388	425	97	235	731
Floating-Point Math	573	762	786	741	1085	2127	1416	629	187	5470	1654
Matrix Multiplication	939	2550	2762	1384	5283	5880	10468	2214	839	2732	2396
Total	3465	6060	6600	5070	16917	18645	21832	6301	2256	12284	7711

Comparatives AVR32-UC3A versus ARM7

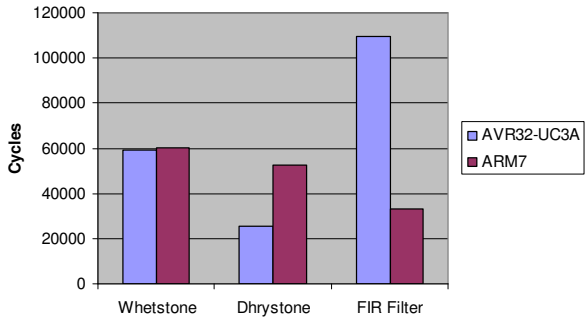
With GCC and IAR

	AVR32-UC3A				ARM7	
	GCC		IAR			
	No opti.	*Full opti.*	*No opti.*	*Full opti.*	*No opti.*	*Full opti.*
8-bit 2-dim Matrix	3207	611	2120	867	2122	475
8-bit Math	206	73	101	33	87	64
8-bit Switch Case	82	17	67	60	51	20
16-bit 2-dim Matrix	3098	611	2090	867	2890	475
16-bit Math	203	76	102	33	102	79
16-bit Switch Case	82	17	67	60	51	20
32-bit Math	186	73	99	33	109	97
Dhrystone	116360	25283 !	114285	52789	83798	52352
FIR Filter	128781	109559	72491	69175	37827	33114
Floating-point Math	850	717	613	573	205	187
Matrix Multiplication	3838	621	2238	939	3424	839
Whetstone	141067	59424 !	144736	143502	61600	60444
Total	**397960**	**197082**	**339009**	**268931**	**192266**	**148166**

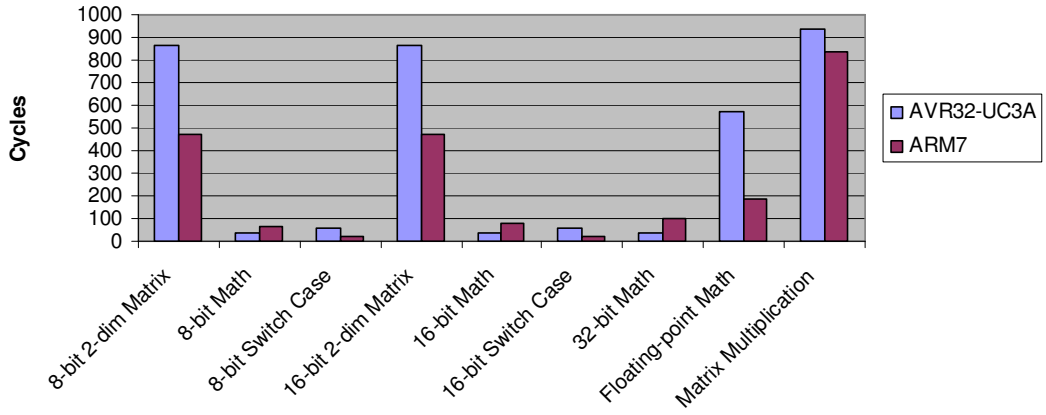
Cycle Count with Optimization for Simple Math Operations
(GCC compiler)



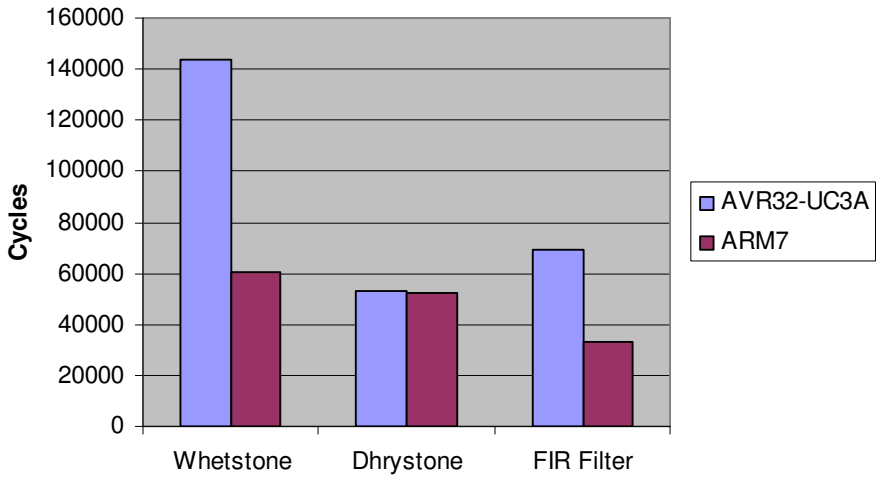
Cycle Count with Optimization for Whetstone, Dhrystone and FIR Filter operation
(GCC compiler)



Cycle Count with Optimization for Simple Math Operations
(IAR compiler)



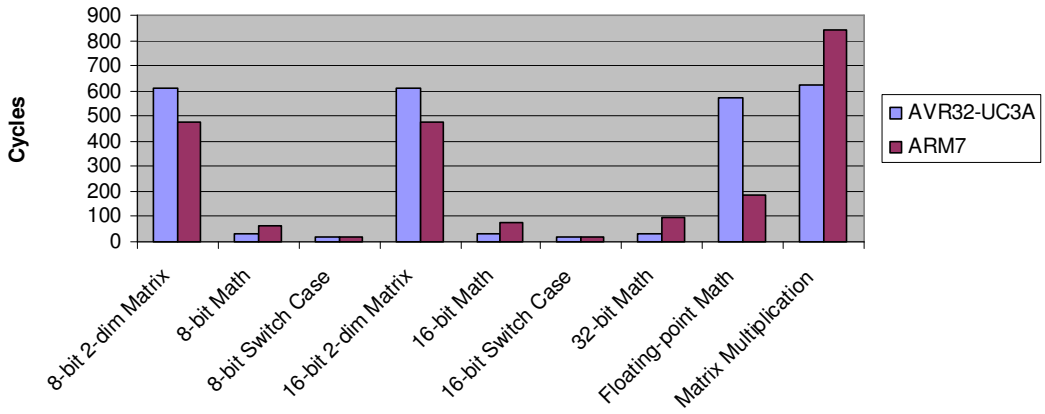
Cycle Count with Optimization for Whetstone, Dhrystone and FIR Filter operation
(IAR compiler)



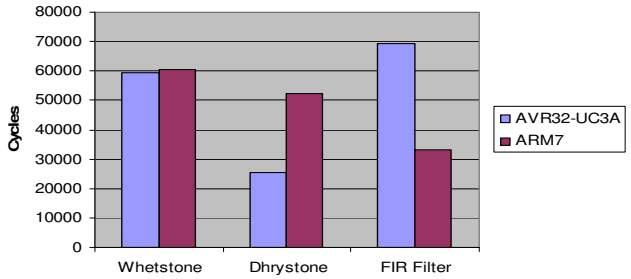
With the best results of GCC and IAR (mixed)

	AVR32-UC3A (best results)		ARM7	
	No opti.	*Full opti.*	*No opti.*	*Full opti.*
8-bit 2-dim Matrix	2120	611	2122	475
8-bit Math	101	33	87	64
8-bit Switch Case	67	17	51	20
16-bit 2-dim Matrix	2090	611	2890	475
16-bit Math	102	33	102	79
16-bit Switch Case	67	17	51	20
32-bit Math	99	33	109	97
Dhrystone	114285	25283 !	83798	52352
FIR Filter	72491	69175	37827	33114
Floating-point Math	613	573	205	187
Matrix Multiplication	2238	621	3424	839
Whetstone	141067	59424 !	61600	60444
Total	**335340**	**156431**	**192266**	**148166**

Cycle Count with Optimization for Simple Math Operations
(Mixed compilers)



Cycle Count with Optimization for Whetstone, Dhrystone and FIR Filter operation
(Mixed compilers)



Benchmark script documentation

TABLE OF CONTENTS

By default, the file "benchmark" is located in the directory "[DSPLIB]/UTILS\SCRIPTS\BENCHMARK" and is set to perform a benchmark on the application located in the directory "[DSPLIB]/EXAMPLES/BENCHMARK/". To try it, open a shell (cygwin) and launch the script "./benchmark". You might have to specify some program paths or directory locations and install specified programs. The script will inform you about requirements.

Script documentation

This script permits to workbench a program, without any intervention. Once it is well configured, it will automatically compile the program to evaluate, download it into the target, retrieves the data and computes them to extract the error information. It will store into an output file the cycle count this process took, the duration in seconds, the error average and the maximal error of the results.
To make this script works you will need a serial cable that you have to connect to the port COM1 of you PC. The other end of this cable has to be connected on the USART of the target.
To configure this script, edit the file "benchmark" and modify the values of the variables defines in the section "BENCH CONFIG".

Applications needed

Cygwin, DataGet, DataExtract, Scilab, Bc, Gawk, Expr, Echo, Cp, Mv, Rm, Make, Rxvt, Sed, sort, Uniq, Printf, Sh, Which.

Constants' configuration

This is the definition of the constants used in the header file generated by this script in order to configure the C program to workbench.

param1_def, param2_def, param3_def, param4_def

Those variables contain the names of the constants.
i.e.: 'VECT2_SIZE'

param1, **param2**, **param3**, **param4**

Those variables are defined by a list containing the values of the actual parameters (param1_def, param2_def, param3_def and param4_def).
The values of the list are separated by the character '#'. If the value has to be a blank, use the special string 'SPACE' to specify this case.
/!\ All the parameters have to be used.
i.e.: '32#64#128#256'

Program's parameters

This is the definition of the parameters that permit computations according to the actual configuration of the C program.

FREQ

This variable defines the main clock frequency of the target. It is used in the time calculation that took the program in its computation. This variable can be dependant of the parameters.
i.e.: If the clock frequency of the target depends on the parameter number 4, you can define it as follow: FREQ='$cur_param4*1000000'.

QA, **QB**

This is the Q formatted parameters that define the actual format to use. The resulting format will be a "Q QA . QB" format.
i.e.: QA='8' and QB='24' that define a 32-bit Q8.24 format number.

OUTPUT_DATA_TYPE

The output data type of the resulting vector that returns the target. Its value must be 'real' or 'imag'.
i.e.: 'real'

Path configuration

Important path used in the script.

CUR_PATH

Specify the directory where is located the Makefile you are using to compile the program (from the script path).
i.e.: '/cygdrive/d/trunk/SERVICES/DSPLIB/EXAMPLES/BENCH/AT32UC3A/GCC'

CONFIG_FILE_PATH

Specify the path of the output header file used to parameter the C program to workbench (from the $CUR_PATH directory).
i.e.: '../../bench_config.h'

SCRIPT_ERROR_FILE_PATH

Specify the path of the Scilab script used to generate the reference output signal of the C program in order to calculate the error (from the $CUR_PATH directory).
i.e.: "`pwd`/RESSOURCES/ref_conv.sce"

Program locations

Specify the programs' path from the $CUR_PATH directory.

DATAGET

Specify the path of the DataGet application.
i.e.: "`pwd`/../../SOURCES/WINDOWS/DATA_GET/DataGet.exe"

DATAEXTRACT

Specify the path of the DataExtract application.
i.e.:"`pwd`/../../SOURCES/WINDOWS/DATA_EXTRACT/DataExtract.exe"

SCILAB_PATH

Specify the path of the Scilab directory.
i.e.: '/cygdrive/c/Program Files/scilab-4.1'

The C program pattern

In order to make the script compatible with the C program, you should use this C program pattern to design your benchmark. This pattern works perfectly with the EVK1100 board.

```
#include <avr32/io.h>
#include "compiler.h"
#include "board.h"

#include "dsp.h"
#include "pm.h"
#include "gpio.h"
#include "usart.h"
#include "count.h"

#include "bench_config.h"

/*! \brief This function is used by the debugging module. It permits to print a string through the USART.
 * \param str The input string to print.
 * \pre It has to be used after having initializing the USART module.
 */
void print_fct(char *str)
{
```

```
	usart_write_line(&AVR32_USART0, str);
}

//! this function intializes the USART module at 9600 bauds
void init_usart()
{
	// Configuration structure for the USART module
	usart_options_t usart_options =
	{
		// Baudrate at 9600 bauds
		.baudrate = 9600,
		// 8 bits per characters
		.charlength = 8,
		// No parity
		.paritytype = USART_NO_PARITY,
		// One stop bit
		.stopbits = USART_1_STOPBIT,
		// Mode normal
		.channelmode = USART_MODE_NORMAL
	};

	// Assign GPIO to USART.
	gpio_enable_module_pin(AVR32_USART0_RXD_0_PIN, AVR32_USART0_RXD_0_FUNCTION);
	gpio_enable_module_pin(AVR32_USART0_TXD_0_PIN, AVR32_USART0_TXD_0_FUNCTION);

	// Initialize USART in RS232 mode.
	usart_init_rs232(&AVR32_USART0, &usart_options, FOSC0);
}

//! The main function
int main(int argc, char *argv[])
{
	unsigned int cycle_count;

	// Swith to external Oscillator 0.
	pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);

	// Initialize the USART
	init_usart();
	// Initialize the DSP debug module
	dsp_debug_initialization(print_fct);

	// Get the actual cycle count
	cycle_count = Get_sys_count();

	// <<< Type here the code you need to workbench >>>

	// Calculate the number of cycles
	cycle_count = Get_sys_count() - cycle_count;

	// Print the number of cycles
	dsp16_debug_printf("Cycle count: %d\n", cycle_count);

	// Print the output
	// <<< use the functions dsp16_debug_print_vect, dsp32_debug_print_vect, dsp16_debug_print_complex_vect or
dsp32_debug_print_complex_vect >>>

	while(1);
}
```

In this program you will have access to the constants you defined in the script file as parameters. Those constants are accessible by including the file “bench_config.h”.
Finally the program has to be compiled defining DSP_BENCH=1 and DSP_DEBUG=1.

The Scilab data comparison script

This is the file that contains the reference signal which should be the result of the output of the program to workbench. It serves to compare this reference signal with the actual output and permits to retrieve the error average and the maximal error of this output signal.

In this file you will just have to generate the signal you should receive with you C program. To do that you will have access to the constants you defined in the script file as parameters. The output signal must be store in the "y" variable and can be a real or a imaginary vector.
Here is an example for the FFT:

```
y_size = 2^NLOG;
t = 0:(y_size-1);
vect2_input = sin(2*%pi*4000*t/40000);

y = fft(vect2_input)/y_size;
```

The output file

The resulting file of this benchmark is a text file containing all the workbench results. Each result is printed on 4 lines: the first line corresponds to the cycle count of the process, the next one is the duration of this process in second, the third is the error average of the result and the last, its maximal error. Then you can easily manage such a file using a spreadsheet.

Why is it not working?

- **The HyperTerminal is opened and so the application DataGet cannot open the COM port.**
  - Close the HyperTerminal and rerun the script.
- **An error is generated by the Scilab data comparison script.**
  - Check the error on the file "dsp_bench_temp" else try to launch this script manually with the required files.
- **An error occurred while compiling the program.**
  - This might be due to an error on your C files. Compile it separately to check the error or look at the file ./dsp_bench_log.txt.